

单片机技术视频大课堂

# AVR 单片机 C 语言轻松学（配视频教程）

严 雨 李 佳 秦文海 编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

ATmega16 是 ATMEL 公司研发的增强型内置 FLASH 的 RISC (Reduced Instruction Set CPU) 精简指令集高速 8 位单片机, 具有体积小、功能强、价格低的特点, 在工业控制、数据采集、智能仪表、机电一体化、家用电器等领域有着广泛的应用, 其应用可以大大提高生产、生活的自动化水平。

本书分为 ATmega16 单片机基础知识、ATmega16 单片机模块应用以及 ATmega16 单片机的应用系统三大部分。本书具有基础内容丰富、循序渐进、由浅入深的特点, 涉及了 ATmega16 单片机从硬件模块基础到软件设计各个方面的知识的特点, 并且基于 Proteus 硬件仿真环境提供了大量的仿真实例, 还提供了 17 个详细讲解的视频供读者深入理解 ATmega16 单片机的使用。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有, 侵权必究。

## 图书在版编目 (CIP) 数据

AVR 单片机 C 语言轻松学: 配视频教程/严雨, 李佳, 秦文海编著. —北京: 电子工业出版社, 2015. 11  
(单片机技术视频大课堂)

ISBN 978-7-121-27372-8

I. ①A… II. ①严… ②李… ③秦… III. ①单片微型计算机-C语言-程序设计 IV. ①TP368.1  
②TP312

中国版本图书馆 CIP 数据核字 (2015) 第 239470 号

策划编辑: 王敬栋

责任编辑: 底 波

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×1092 1/16 印张: 19.25 字数: 492.8 千字

版 次: 2015 年 11 月第 1 版

印 次: 2015 年 11 月第 1 次印刷

印 数: 3 000 册 定价: 58.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlt@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

# 前 言

---

## 行业背景

ATmega16 是 ATMEL 公司研发的增强型内置 FLASH 的 RISC (Reduced Instruction Set CPU) 精简指令集高速 8 位单片机 (AVR, 其得名于设计师 A 先生和 V 先生), 具有体积小、功能强、价格低的特点, 在工业控制、数据采集、智能仪表、机电一体化、家用电器等领域有着广泛的应用, 其应用可以大大提高生产、生活的自动化水平。

## 关于本书

本书基于 ICCAVR 集成开发环境和 Proteus 硬件仿真环境分章节介绍了 ATmega16 单片机的基础构成、内部资源以及外部器件的使用方法, 包括其体系结构、C 语言、定时计数器内部资源以及 LED、独立按键、继电器等外部资源。

本书提供了 ATmega16 单片机的多个应用实例, 在 Proteus 中读者观察到这些应用实例的仿真执行情况。本书还制作了 17 个和章节内容对应的讲解视频, 以便于读者更好地理解 ATmega16 单片机的使用。

本书分为 ATmega16 单片机基础知识、ATmega16 单片机模块应用以及 ATmega16 单片机的应用系统三大部分。

- ATmega16 单片机基础知识: 第 1 章至第 4 章, 介绍了 AVR 系列单片机的内部结构、C 语言、ICCAVR 集成开发环境的使用方法以及 Proteus 硬件仿真环境的使用方法。
- ATmega16 单片机模块应用: 第 5 章至第 10 章, 介绍了 ATmega16 单片机的外部引脚、外部中断、定时计数器、串口、SPI 和 TWI 接口、模拟比较器和 ADC 模块以及看门狗和内部 E<sup>2</sup>PROM 的使用方法。
- ATmega16 单片机的应用系统: 第 11 章, 介绍了单引脚扩展多按键、简易电子琴以及商场灯光控制系统三个基于 ATmega16 的实际应用系统。

本书提供的视频内容及其长度说明如下。

- 【视频 1】ICCAVR 的基础使用方法 (32 分钟)。
- 【视频 2】Proteus 的基础使用方法 (30 分钟)。
- 【视频 3】Proteus 中的 ATmega16 (13 分钟)。
- 【视频 4】Proteus 和 ICCAVR 的联合使用 (13 分钟)。
- 【视频 5】ATmega16 的外部引脚和 Proteus 中的示波器使用 (15 分钟)。
- 【视频 6】发光二极管 LED 的应用 (17 分钟)。



- 【视频 7】单位数码管的应用 (12 分钟)。
- 【视频 8】按键和行列扫描键盘的应用 (20 分钟)。
- 【视频 9】ATmega16 的外部中断及其应用 (8 分钟)。
- 【视频 10】ATmega16 的定时计数器应用 (20 分钟)。
- 【视频 11】ATmega16 单片机的串口及其应用 (26 分钟)。
- 【视频 12】ATmega16 单片机的比较器及其应用 (12 分钟)。
- 【视频 13】ATmega16 单片机的 ADC 及其应用 (20 分钟)。
- 【视频 14】ATmega16 单片机的看门狗和 E<sup>2</sup>PROM 的使用方法及其应用 (12 分钟)。
- 【视频 15】单 I/O 引脚扩展多按键应用系统 (16 分钟)。
- 【视频 16】简易电子琴应用系统 (24 分钟)。
- 【视频 17】商场灯光控制系统 (25 分钟)。

### 本书特色

- 基础内容丰富、循序渐进、由浅入深, 涉及了 ATmega16 单片机从硬件模块基础到软件设计各个方面的知识。
- 基于 Proteus 硬件仿真环境提供了大量仿真实例。
- 提供了 17 个详细讲解的视频供读者深入理解 ATmega16 单片机的使用。

### 作者介绍

本书由严雨、李佳、秦文海编著。参与本书编写的还有李若谷、刘洋洋、王闯、严安国、何世兰、姚宗旭、葛祥磊、徐慧超、张玉梅、夏宁和韩敏等人。在此, 对以上人员致以诚挚的谢意。由于时间仓促, 程序较多, 受学识水平所限, 错误之处在所难免, 请广大读者给予批评指正。



# 目 录

<b>第1章</b>	<b>ATmega16 单片机基础</b>	1
1.1	AVR 系列单片机	1
1.2	ATmega16 单片机的特点、硬件结构和封装	1
1.3	ATmega16 单片机的内核	4
1.3.1	算术逻辑单元 ALU	5
1.3.2	状态寄存器 SREG	5
1.3.3	通用寄存器	5
1.3.4	堆栈	6
1.3.5	中断和复位处理模块	7
1.4	ATmega16 单片机的存储器体系	8
1.4.1	程序存储器	8
1.4.2	数据存储器	8
1.4.3	E <sup>2</sup> PROM 存储器	9
1.5	ATmega16 单片机的系统时钟	10
1.5.1	ATmega16 的系统时钟组成	10
1.5.2	ATmega16 的时钟源选择	11
1.5.3	晶体振荡器	11
1.5.4	低频晶体振荡器	12
1.5.5	外部 RC 振荡器	12
1.5.6	片内 RC 振荡器	13
1.5.7	外部时钟源	14
1.6	ATmega16 单片机的电源管理	15
1.7	ATmega16 单片机的复位	17
1.7.1	ATmega16 的复位源	17
1.7.2	上电复位	18
1.7.3	外部复位	19
1.7.4	掉电检测复位	19
1.7.5	看门狗复位	19
1.7.6	ATmega16 的复位控制寄存器	20
1.7.7	片内基准电压	20
1.8	ATmega16 单片机的中断系统	21
<b>第2章</b>	<b>ATmega16 单片机的指令和 C 语言</b>	23
2.1	ATmega16 单片机的指令系统	23





2.1.1	ATmega16 单片机的指令集 .....	23
2.1.2	ATmega16 单片机的寻址方式 .....	27
2.2	ATmega16 单片机 C 语言的数据类型、运算符和表达式 .....	28
2.2.1	常量和变量 .....	28
2.2.2	算术运算、赋值、逻辑运算以及关系运算 .....	28
2.2.3	自增减、复合和逗号运算 .....	29
2.2.4	位运算 .....	30
2.2.5	运算的优先级 .....	30
2.3	ATmega16 单片机 C 语言的结构 .....	31
2.4	ATmega16 单片机 C 语言的函数 .....	32
2.4.1	函数的定义、参数和返回值 .....	32
2.4.2	函数的调用 .....	32
2.4.3	局部变量和全局变量 .....	32
2.5	ATmega16 单片机 C 语言的数组和指针 .....	33
2.6	ATmega16 单片机 C 语言的自构造类型 .....	34
2.6.1	结构体 .....	34
2.6.2	联合体 .....	35
2.6.3	枚举 .....	36
<b>第 3 章</b>	<b>ATmega16 单片机的 ICC AVR 软件开发环境 .....</b>	<b>37</b>
3.1	ATmega16 单片机的软件开发环境 .....	37
3.2	安装 ICC AVR .....	37
3.3	ICC AVR 的工作界面 .....	39
3.4	ICC AVR 的菜单栏和快捷工具栏 .....	40
3.4.1	ICC AVR 的菜单栏 .....	40
3.4.2	ICC AVR 的快捷工具栏 .....	44
3.5	ICC AVR 的扩展关键字 .....	45
3.5.1	中断关键字 .....	45
3.5.2	非挥发寄存器关键字 .....	46
3.5.3	数据段关键字 .....	46
3.6	ICC AVR 的文件 .....	46
3.6.1	ICC AVR 的常用文件类型 .....	46
3.6.2	ICC AVR 的库函数文件 .....	47
3.6.3	ICC AVR 的启动文件 .....	47
3.7	“Hello World!”——ICC AVR 的应用实例 .....	48
<b>第 4 章</b>	<b>ATmega16 单片机的硬件开发和 Proteus 硬件仿真环境 .....</b>	<b>52</b>
4.1	ATmega16 单片机的硬件系统开发流程和开发工具 .....	52
4.1.1	ATmega16 单片机的硬件系统开发流程 .....	52
4.1.2	ATmega16 单片机的硬件开发工具 .....	53
4.2	Proteus 应用基础 .....	54



4.2.1	Proteus 的界面和支持的文件 .....	54
4.2.2	Proteus 的菜单 .....	56
4.2.3	Proteus 的快捷工具栏和工具箱 .....	67
4.3	Proteus 的使用流程 .....	70
4.4	Proteus 中的 ATmega16 及其使用 .....	70
4.5	Proteus 和 ICC AVR 联合使用 .....	72
<b>第5章</b>	<b>ATmega16 单片机的 I/O 引脚和外部中断 .....</b>	<b>78</b>
5.1	ATmega16 外部引脚基础使用方法 .....	78
5.1.1	ATmega16 的 I/O 引脚的结构 .....	78
5.1.2	ATmega16 的 I/O 引脚配置 .....	79
5.1.3	ATmega16 的 I/O 引脚电平读取 .....	81
5.1.4	ATmega16 的 I/O 引脚低功耗处理 .....	82
5.2	ATmega16 外部引脚的第二功能 .....	82
5.3	ATmega16 的外部中断 .....	86
5.3.1	MCU 控制寄存器 (MCUCR) .....	86
5.3.2	MCU 控制与状态寄存器 (MCUCSR) .....	87
5.3.3	通用中断控制寄存器 (GICR) .....	87
5.3.4	通用中断标志寄存器 (GIFR) .....	88
5.4	ATmega16 的 I/O 引脚和中断的应用实例 .....	88
5.4.1	I/O 引脚输出高低脉冲电平实例 .....	88
5.4.2	I/O 引脚驱动发光二极管 (LED) 实例 .....	92
5.4.3	I/O 引脚驱动单位数码管实例 .....	98
5.4.4	I/O 引脚驱动独立按键实例 .....	104
5.4.5	I/O 引脚驱动行列键盘实例 .....	110
5.4.6	外部中断控制 I/O 引脚输出实例 .....	115
<b>第6章</b>	<b>ATmega16 单片机的定时计数器 .....</b>	<b>119</b>
6.1	定时计数器 T/C0 .....	119
6.1.1	T/C0 的相关寄存器 .....	119
6.1.2	T/C0 的工作模式 .....	123
6.2	定时计数器 T/C1 .....	127
6.2.1	T/C1 的相关寄存器 .....	127
6.2.2	T/C1 的工作模式 .....	132
6.3	定时计数器 T/C2 .....	137
6.3.1	T/C2 的相关寄存器 .....	138
6.3.2	T/C2 的工作模式 .....	141
6.4	ATmega16 的定时计数器的应用实例 .....	144
6.4.1	T/C0 控制 I/O 引脚输出方波 .....	144
6.4.2	T/C1 控制 I/O 引脚输出 PWM .....	147
6.4.3	外部晶体秒定时 .....	149



<b>第 7 章</b>	<b>ATmega16 单片机的串口</b>	153
7.1	ATmega16 串口的结构	153
7.2	ATmega16 串口的寄存器	154
7.2.1	串口数据寄存器 (UDR)	154
7.2.2	串口控制和状态寄存器 A (UCSRA)	154
7.2.3	串口控制和状态寄存器 B (UCSRB)	155
7.2.4	串口控制和状态寄存器 C (UCSRC)	156
7.2.5	串口波特率寄存器 (UBRRH 和 UBRRL)	157
7.3	ATmega16 串口的使用方法	160
7.3.1	选择 ATmega16 串口的时钟源	160
7.3.2	选择 ATmega16 串口的数据帧格式	162
7.3.3	ATmega16 串口的数据收发	163
7.3.4	ATmega16 串口的多机通信	165
7.4	ATmega16 串口的应用实例	166
7.4.1	ATmega16 串口数据发送	166
7.4.2	和 PC 进行串行通信	172
<b>第 8 章</b>	<b>ATmega16 单片机的 TWI 和 SPI 总线接口</b>	178
8.1	TWI 总线基础	178
8.1.1	TWI 总线的数据交互过程	178
8.1.2	TWI 总线的地址	180
8.2	TWI 总线模块相关寄存器	181
8.2.1	比特率控制寄存器 (TWBR)	181
8.2.2	TWI 控制寄存器 (TWCR)	181
8.2.3	TWI 状态寄存器 (TWSR)	182
8.2.4	TWI 数据寄存器 (TWDR)	183
8.2.5	TWI 从机地址寄存器 (TWAR)	183
8.3	TWI 总线模块的使用	183
8.4	TWI 总线模块的数据传输方式	185
8.4.1	主机发送模式 (MT)	185
8.4.2	主机接收模式 (MR)	186
8.4.3	从机发送模式 (ST)	187
8.4.4	从机接收模式 (SR)	188
8.5	TWI 总线的仲裁	189
8.6	SPI 总线基础	189
8.7	SPI 总线模块相关寄存器	191
8.7.1	SPI 控制寄存器 SPCR	191
8.7.2	SPI 状态寄存器 SPSR	192
8.7.3	SPI 数据寄存器 SPDR	193
8.8	SPI 总线接口的工作模式	193





8.9	TWI 和 SPI 总线模块应用实例 .....	195
8.9.1	ATmega16 双机使用 TWI 总线模块进行通信 .....	195
8.9.2	ATmega16 双机使用 SPI 总线模块进行通信 .....	203
<b>第9章</b>	<b>ATmega16 单片机的比较器和 ADC 模块 .....</b>	<b>211</b>
9.1	ATmega16 单片机的比较器 .....	211
9.1.1	模拟比较器基础 .....	211
9.1.2	模拟比较器的寄存器 .....	211
9.1.3	模拟比较器的输入通道 .....	213
9.2	ATmega16 单片机的 ADC 模块 .....	213
9.2.1	ADC 模块基础 .....	214
9.2.2	ADC 模块的寄存器 .....	215
9.2.3	ADC 模块的转换过程 .....	219
9.2.4	ADC 模块的输入通道和参考电源 .....	221
9.2.5	ADC 模块的转换结果和精度定义 .....	222
9.3	ATmega16 比较器的应用实例 .....	224
9.3.1	双通道模拟信号比较应用实例 .....	224
9.3.2	多通道模拟信号比较应用实例 .....	227
9.4	ATmega16 ADC 模块的应用实例 .....	231
9.4.1	单通道模拟信号采集实例 .....	231
9.4.2	多通道模拟信号采集实例 .....	234
9.4.3	增益放大模拟信号采集实例 .....	237
9.4.4	差分模拟信号比较采集实例 .....	241
<b>第10章</b>	<b>ATmega16 的其他内部资源 .....</b>	<b>245</b>
10.1	看门狗 (WDT) .....	245
10.1.1	看门狗基础 .....	245
10.1.2	看门狗的寄存器 .....	245
10.1.3	看门狗的启动和关闭 .....	246
10.2	内部 E <sup>2</sup> PROM .....	246
10.2.1	E <sup>2</sup> PROM 的操作 .....	247
10.2.2	E <sup>2</sup> PROM 的寄存器 .....	248
10.2.3	E <sup>2</sup> PROM 的操作函数 .....	249
10.3	内置看门狗和 E <sup>2</sup> PROM 应用实例 .....	250
10.3.1	内置看门狗模块测试应用实例 .....	250
10.3.2	E <sup>2</sup> PROM 读写应用实例 .....	254
<b>第11章</b>	<b>ATmega16 的应用系统 .....</b>	<b>258</b>
11.1	单 I/O 引脚扩展多按键 .....	258
11.1.1	应用系统背景 .....	258
11.1.2	设计思路 .....	258



11.1.3	硬件系统设计 .....	259
11.1.4	软件系统设计 .....	261
11.1.5	应用系统的仿真和总结 .....	263
11.2	简易电子琴 .....	265
11.2.1	应用系统背景 .....	265
11.2.2	设计思路 .....	266
11.2.3	硬件系统设计 .....	267
11.2.4	软件系统设计 .....	271
11.2.5	应用系统的仿真和总结 .....	277
11.3	商场灯光控制 .....	279
11.3.1	应用系统背景 .....	279
11.3.2	设计思路 .....	279
11.3.3	硬件系统设计 .....	279
11.3.4	软件系统设计 .....	288
11.3.5	应用系统的仿真和总结 .....	298



# 第1章 ATmega16 单片机基础

ATmega16 是 ATMEL 公司研发的增强型内置 FLASH 的 RISC (Reduced Instruction Set CPU) 精简指令集高速 8 位单片机 (AVR, 其得名于设计师 A 先生和 V 先生), 本章介绍 AVR 系列单片机的特点和 ATmega16 单片机的一些基础知识, 包括内核、存储器体系、系统时钟、电源管理、复位、外部引脚封装和中断系统等。

## 1.1 AVR 系列单片机

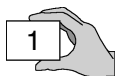
单片机是单片微型计算机的简称, 是一种将运算控制器、存储器、寄存器 I/O 接口以及一些常用的功能模块都集成到一块芯片上的计算机, 常常用于工业控制, 小型家电等需要嵌入式控制的场合, 其按照结构可以分为 AVR 系列、51 系列、PIC 系列等, 其中 AVR 系列多用于工业控制, 它具有以下的特点。

- 采用哈佛结构, 具备 1MIPS /MHz 的高速运行处理能力。
- 支持超功能精简指令集, 具有 32 个通用工作寄存器, 避免了 51 等系列单片机由于采用单个 ACC 进行处理造成的瓶颈现象。
- 内置快速的存取寄存器组、有单周期指令系统, 大大优化了目标代码的大小、执行效率高。
- 外部引脚驱动能力大, 作为输出时可输出 40mA 的大电流, 作为输入时可设置为三态高阻抗输入或带上拉电阻输入, 具备 10 ~ 20mA 灌电流的能力。
- 片内集成了多种频率的 RC 振荡器, 具有上电自动复位、看门狗、启动延时等功能, 使得外围电路更加简单, 系统更加稳定可靠。
- 片内资源丰富, 通常集成了 E<sup>2</sup>PROM、PWM、RTC、SPI、UART、TWI、ISP、ADC、比较器、WDT 等功能, 可以减少系统中外围器件的数量, 有效地实现“单片”系统。
- 支持 ISP 和 IAP 功能, 方便下载程序, 并且支持 JTAG 的片上调试。

## 1.2 ATmega16 单片机的特点、硬件结构和封装

ATmega16 是基于 RISC 结构的 8 位高性能、低功耗的处理器, 是 AVR 单片机的一种, 其主要特点如下。

- 支持 131 条 AVR 指令, 其中大多数指令执行时间为单个时钟周期, 执行速度快。
- 内部有 32 个 8 位通用工作寄存器, 支持全静态工作。
- 当处理器工作于 16MHz 工作频率时性能高达 16MIPS。
- 硬件乘法器只需两个时钟周期。



- 内置 1KB 的片内 SRAM，16KB 的系统内可编程 FLASH，512B 的 E<sup>2</sup>PROM。
- 内置具有独立锁定位的可选 Boot 代码区，并且可以通过片上 Boot 程序实现系统内编程。
- 支持符合 JTAG 标准的边界扫描，提供和 IEEE 1149.1 标准兼容的 JTAG 硬件接口。
- 内置两个具有独立预分频和比较功能的 8 位定时计数器，内置一个具有预分频、比较功能和捕捉功能的 16 位定时计数器，内置一个具有独立振荡器的实时计数器 RTC，这些定时计数器可以提供四通道 PWM。
- 内置片内模拟比较器和 8 通道 10 位 ADC，可以组合为 8 个单端通道或者 7 组差分通道，其中有 2 个具有可编程增益（1×、10×或 200×）的差分通道。
- 内置多种串行通信接口，包括 TWI（I<sup>2</sup>C）两线接口、可编程串口（USART）、可工作于主机/从机模式的 SPI 串行接口。
- 内置具有独立片内振荡器的可编程看门狗定时器。
- 可以提供上电复位以及可编程的掉电检测，其内置经过标定的 RC 振荡器，可以不需要外部时钟工作，并且内置了片外中断源，提供空闲模式、省电模式等 6 种工作模式。
- 提供 32 个可编程的 I/O 口，支持 40 引脚 PDIP 封装，44 引脚 TQFP 封装，44 引脚 MLF 封装。
- 支持 2.7 ~ 5.5V（ATmega16L，工作频率 0 ~ 8MHz）和 4.5 ~ 5.5V（ATmega16，工作频率 0 ~ 16MHz）工作电压。

图 1.1 是 ATmega16 的内部结构示意图，它由 ALU、通用寄存器、程序存储器、数据 RAM、中断模块和内部扩展资源等组成，各部分详细说明如下。

- ALU（运算器）。支持通用寄存器之间以及通用寄存器和常数之间的算术和逻辑运算，ALU 也可以执行单寄存器操作，当运算完成之后更新相应的状态寄存器的内容以反映操作结果。ATmega16 通过有条件或者无条件的跳转指令和调用指令来控制程序的工作流程，从而可以直接寻址整个地址空间。
- 通用寄存器。其中 6 个寄存器可以联合起来构成 3 个 16 位的 X、Y、Z 间接寻址寄存器，可以用来寻址数据空间以实现高效的地址运算，其中一个指针还可以作为程序存储器查询表的地址指针。
- 程序存储器。这是可以在线编程的 FLASH，其快速访问寄存器由 32 个 8 位通用寄存器组成，访问时间为一个时钟周期，从而 ATmega16 可以实现单时钟周期的运算器操作，在典型的运算器操作中，两个分别位于不同通用寄存器中的操作数被同时访问，然后执行运算，结果再被送回到通用寄存器，整个指令执行过程仅需一个时钟周期。
- 中断模块。ATmega16 内置一个灵活的中断模块，其控制寄存器位于 I/O 空间内，并且有一个位于状态寄存器 SREG 中的全局中断使能位，每个中断在中断向量表里都有独立的中断向量，其优先级与该中断向量在中断向量表的位置有关，中断向量地址越低，其优先级越高。
- 内部扩展资源。ATmega16 内部集成了多种外部资源，包括 SPI 接口、ADC 接口、E<sup>2</sup>PROM、串行模块等。



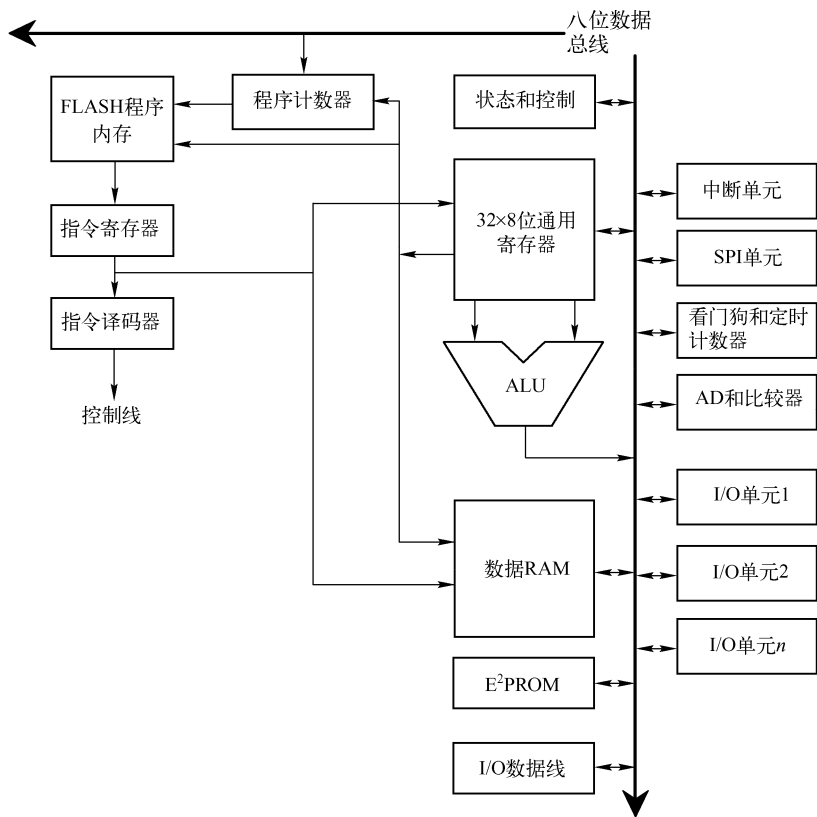
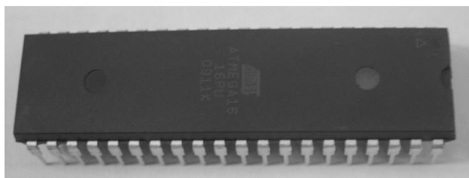


图 1.1 ATmega16 的内部结构示意图

ATmega16 有 DIP-40 和 TQFP/MLF-44 两种封装形式，其实物和引脚封装模型分别如图 1.2 和图 1.3 所示，其外部引脚说明如下。

DIP封装的ATmega16



TQFP封装的ATmega16



图 1.2 ATmega16 实物

- VCC：电源正输入引脚。
- GND：电源地输入引脚。
- PA7 ~ PA0：外部引脚端口 A，其为 8 位双向输入/输出端口，具有可编程的内部上拉电阻，其输出缓冲器具有对称的驱动特性，可以输出和吸收较大电流；当作为输入使用时，若内部上拉电阻使能，端口被外部电路拉低时将输出电流；在复位过程中，即使系统时钟还未起振，端口 A 也处于高阻状态；其第二功能为 ADC 转换器的模拟输入端。

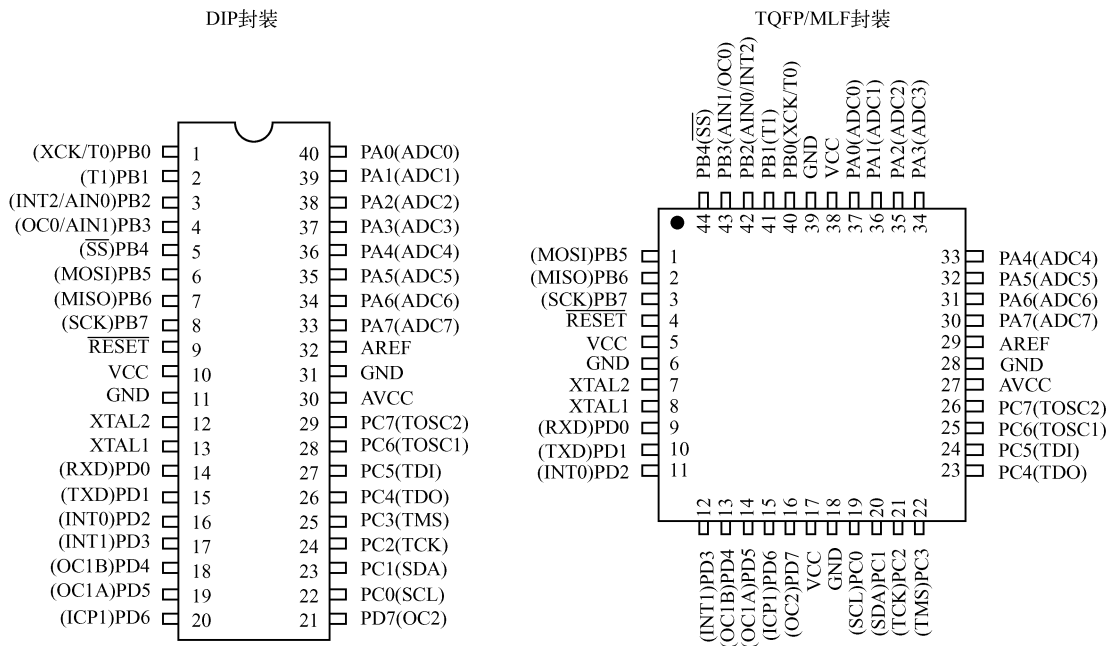


图 1.3 ATmega16 引脚封装

- PB7 ~ PB0: 外部引脚端口 B, 也为 8 位双向输入/输出端口, 其特点和端口 A 类似; 端口 B 也可以用作其他不同的特殊功能。
- PC7 ~ PC0: 外部引脚端口 C, 同样也为 8 位双向输入/输出端口, 其特点和端口 A、端口 B 类似; 需要注意的是, 如果 JTAG 接口被使能, 即使复位出现引脚 PC5 (TDI)、PC3 (TMS) 与 PC2 (TCK) 的上拉电阻被激活, 除去移出数据的 TAP 态外, TD0 引脚均为高阻态。
- PD7 ~ PD0: 外部引脚端口 D, 同样也是 8 位双向输入输出端口, 其特点和端口 A、端口 B、端口 C 类似。
- RESET: 复位输入引脚, 在该引脚上加上持续时间超过最小门限时间的低电平将引起系统复位。
- XTAL1: 反向振荡放大器与片内时钟操作电路的输入端。
- XTAL2: 反向振荡放大器的输出端。
- AVCC: 内置 ADC 转换器的参考电源, 不使用 ADC 时, 该引脚应直接与 VCC 引脚连接, 当使用 ADC 时该引脚应该通过一个低通滤波电路与 VCC 引脚连接。
- AREF: ADC 转换器的模拟基准输入引脚。

### 1.3 ATmega16 单片机的内核

ATmega16 单片机的内核由算术逻辑单元、状态寄存器、通用寄存器、堆栈、中断和复位处理模块组成。



### 1.3.1 算术逻辑单元 ALU

算术逻辑单元 (ALU) 是 ATmega16 内核中执行各种算术和逻辑运算操作的部件, 其基本操作包括加、减、乘、除四则运算 (算术运算), 与、或、非、异或等逻辑操作 (逻辑运算), 以及移位、比较和传送等操作。ALU 与 32 个通用工作寄存器直接相连, ATmega16 的寄存器与寄存器之间、寄存器与立即数之间的 ALU 运算只需要一个时钟周期。ALU 的操作分为三类: 算术、逻辑和位操作, 此外 ALU 还能支持无/有符号数和分数乘法。

### 1.3.2 状态寄存器 SREG

ATmega16 的状态寄存器包含了最近执行的算术指令的相关结果信息, 用户可以根据这些信息来实现条件操作, 状态寄存器 SREG 的内部结构如表 1.1 所示, 其详细说明如下。

表 1.1 ATmega16 的状态寄存器 SREG

BIT	I	T	H	S	V	N	Z	C
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- **I**: 全局中断使能位, 当 I 被置位时使能全局中断, 单独的中断使能由其他独立的控制寄存器控制; 如果 I 被清零, 则不论单独中断标志置位与否, 都不会产生中断。任意一个中断发生后 I 将被清零, 而执行 RETI (中断服务程序退出) 指令后 I 恢复置位以使能中断, I 也可以通过 SEI (置位全局中断使能位) 和 CLI (清除全局中断使能位) 指令来置位和清零。
- **T**: 位复制存储位, 位复制指令 BLD 和 BST 利用 T 作为目的或源地址, BST 指令把寄存器的某一位复制到 T, 而 BLD 把 T 复制到寄存器的某一位。
- **H**: 半进位标志位, H 被置位时表示算术操作发生了半进位, 此标志对于 BCD 运算非常有用。
- **S**: 负数标志位, 用于存放 N 与 2 的补码和溢出标志 V 的异或结果。
- **V**: 补码溢出标志, 支持二进制补码运算。
- **N**: 负数标志位, 表明算术或逻辑操作结果为负。
- **Z**: 零标志位, 表明算术或逻辑操作结果为零。
- **C**: 进位标志位, 表明算术或逻辑操作发生了进位。

### 1.3.3 通用寄存器

ATmega16 有 32 个通用寄存器, 其针对 ATmega16 的指令集进行了优化, 支持以下的输入/输出方案。

- 输入为一个 8 位操作数, 输出一个 8 位结果。
- 输入为两个 8 位操作数, 输出一个 8 位结果。
- 输入为两个 8 位操作数, 输出一个 16 位结果。
- 输入为一个 16 位操作数, 输出一个 16 位结果。



图 1.4 是 ATmega16 的通用寄存器结构示意图，每个通用寄存器都有一个对应的地址，将它们直接映射到用户数据空间的前 32 个地址，X、Y、Z 寄存器可以设置为指向任意寄存器的指针。

	7	0	Addr.	
通用 工作 寄存器	R0		\$00	
	R1		\$01	
	R2		\$02	
	...			
	R13		\$0D	
	R14		\$0E	
	R15		\$0F	
	R16		\$10	
	R17		\$11	
	...			
	R26		\$1A	X寄存器, 低字节
	R27		\$1B	X寄存器, 高字节
	R28		\$1C	Y寄存器, 低字节
	R29		\$1D	Y寄存器, 高字节
	R30		\$1E	Z寄存器, 低字节
	R31		\$1F	Z寄存器, 高字节

图 1.4 ATmega16 的通用寄存器结构示意图

通用寄存器 R26 ~ R31 除了用作通用寄存器外，还可以作为数据间接寻址用的地址指针 X、Y、Z，如图 1.5 所示。在不同的寻址模式中，这些地址寄存器可以实现固定偏移量，自动加 1 和自动减 1 功能。

X寄存器	15	XH		XL	0
	7		0	7	0
	R27(\$1B)			R26(\$1A)	
Y寄存器	15	YH		YL	0
	7		0	7	0
	R29(\$1D)			R28(\$1C)	
Z寄存器	15	ZH		ZL	0
	7		0	7	0
	R31(\$1F)			R30(\$1E)	

图 1.5 X、Y、Z 寄存器

### 1.3.4 堆栈

ATmega16 的堆栈主要用来保存临时数据、局部变量、子程序和中断子程序的返回地址，堆栈指针总是指向堆栈的顶部。



#### 注意

ATmega16 的堆栈是向下生长的，即当有新数据被推入堆栈时，堆栈指针的数值将减小。

ATmega16 的堆栈指针指向位于 SRAM 的函数或者中断堆栈，在调用子程序和使能中断之前必须先初始化堆栈，且堆栈指针必须指向高于 0x60 的地址空间，堆栈的详细增减说明如下。





- 当使用 PUSH 指令将数据推入堆栈时堆栈指针减 1。
- 当子程序或中断返回地址被推入堆栈时指针将减 2。
- 使用 POP 指令将数据弹出堆栈时，堆栈指针加 1。
- 使用 RET 或 RETI 指令从子程序或中断返回时堆栈指针加 2。

ATmega16 的堆栈指针实质上是 I/O 空间中的两个 8 位寄存器，如表 1.2 所示，可以分为 SPH (SP8 ~ SP15) 和 SPL (SP0 ~ SP7) 两部分。

表 1.2 ATmega16 的堆栈指针寄存器

BIT	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

### 1.3.5 中断和复位处理模块

ATmega16 有不同的中断源，每个中断和复位在程序空间都有独立的中断向量，所有的中断事件都有自己的使能位，当使能位被置位，且状态寄存器的全局中断使能位 I 也被置位时，产生中断事件。根据程序计数器 PC 的不同，在引导锁定位 BLB02 或 BLB12 被编程的情况下，中断可能被自动禁止，这个特性提高了软件的安全性。

程序存储区的最低地址默认为复位向量和中断向量，完整的向量列表参考表 1.18。该表列出了不同中断的优先级，向量所在的地址越低，优先级越高。RESET 具有最高的优先级，第二个为 INTO（外部中断请求 0）。通过置位通用中断控制寄存器（GICR）的 IVSEL，中断向量可以移至引导 FLASH 的起始处，编程熔丝位 BOOTRST 也可以将复位向量移至引导 FLASH 的起始处。

在任意一个中断发生时，全局中断使能位 I 被清零，从而禁止了所有其他的中断，用户软件可以在中断程序里置位 I 来实现中断嵌套，此时所有的中断都可以中断当前的中断服务程序在执行 RETI 指令后 I 自动置位。

ATmega16 有两种类型的中断。第一种类型的中断是由事件触发并置位中断标志位。对于这些中断，程序计数器跳转到实际的中断向量以执行中断处理程序，同时硬件将清除相应的中断标志，中断标志位也可以通过对其写“1”的方式来清除，当中断发生后，如果相应的中断使能位为“0”，则中断标志位置位，并一直保持到中断执行，或者被软件清除。类似的，如果全局中断标志被清零，则所有已发生的中断都不会被执行，直到 I 被置位。然后挂起的各个中断按中断优先级依次执行。第二种类型的中断则是只要中断条件满足，就会一直触发，这些中断不需要中断标志位，若中断条件在中断使能之前就消失了，中断不会被触发。

ATmega16 在退出中断后总是回到主程序后并至少执行一条指令才可以去执行其他被挂起的中断。需要注意的是，进入中断服务程序时，状态寄存器不会自动保存，中断返回时也



不会自动恢复。这些工作必须由用户通过软件来完成。当使用 CLI 指令来禁止中断时，中断禁止立即生效，没有中断可以在执行 CLI 指令后发生，即使它是在执行 CLI 指令的同时发生的。

ATmega16 的中断响应操作最少为 4 个时钟周期，在 4 个时钟周期后，程序跳转到实际的中断处理例程。在这 4 个时钟周期期间 PC 将自动入栈，在通常情况下，中断向量为一个跳转指令，此跳转需要 3 个时钟周期，如果中断在一个多时钟周期指令执行期间发生，则在此多周期指令执行完毕后 ATmega16 才会执行中断程序。若中断发生时 ATmega16 处于休眠模式，中断响应时间还需增加 4 个时钟周期。此外，还要考虑不同的休眠模式所需要的启动时间。

ATmega16 的中断返回操作也需要 4 个时钟周期，在此期间 PC（两个字节）将被弹出栈，堆栈指针加 2，状态寄存器 SREG 的 I 位被置位。

## 1.4 ATmega16 单片机的存储器体系

ATmega16 单片机的存储器空间由程序存储器（FLASH）、数据存储器（SDRAM）和 E<sup>2</sup>PROM 存储器组成，它们是相互独立的。

### 1.4.1 程序存储器

ATmega16 具有 16KB 内部 FLASH 用于存放程序指令代码，支持可在线编程（ISP）和在应用编程（IAP），因为 ATmega16 的所有指令均为 16 位或 32 位，所以 FLASH 被组织成 8K×16 位的形式，并且被分为引导程序区（BOOT）和应用程序区两个不同的区，如图 1.6 所示。



图 1.6 ATmega16 的程序存储区

### 1.4.2 数据存储器

图 1.7 是 ATmega16 的数据存储器内部地址结构，前 1120 个字节的数据存储器包括了通用寄存器、I/O 存储器及内部数据 SRAM，起始的 96 个地址用于通用寄存器与 64 个 I/O 存储器，接着是 1024 字节的内部数据 SRAM。



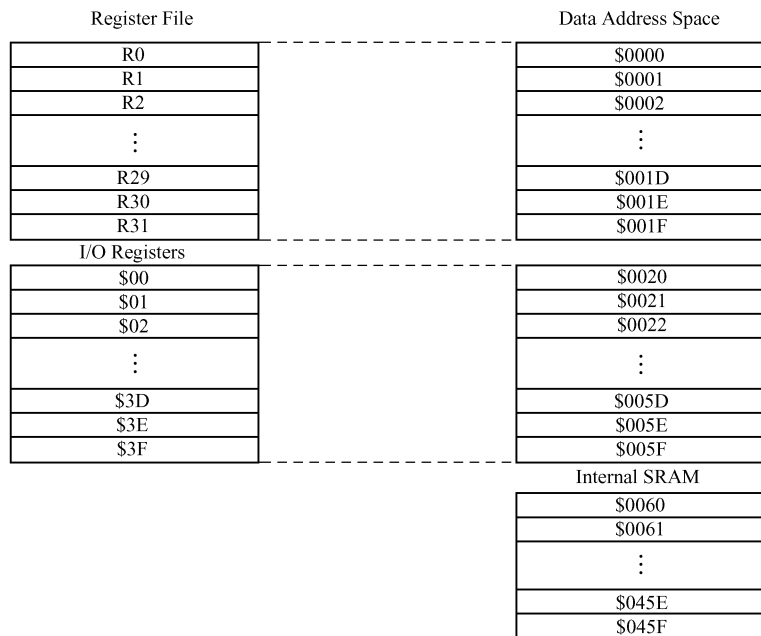


图 1.7 ATmega16 的内部数据存储器结构

数据存储器的寻址方式分为 5 种：直接寻址、带偏移量的间接寻址、间接寻址、带预减量的间接寻址和带后增量的间接寻址。寄存器文件中的寄存器 R26 到 R31 为间接寻址的指针寄存器；直接寻址范围可达整个数据区；带偏移量的间接寻址模式则能够寻址到由寄存器 Y 和 Z 给定的基址附近的 63 个地址。在自动预减和后加的间接寻址模式中，寄存器 X、Y 和 Z 自动增加或减少。

ATmega16 的全部 32 个通用寄存器、64 个 I/O 寄存器及 1024 个字节的内部数据 SRAM 可以通过所有上述的寻址模式进行访问。

ATmega16 所有的 I/O 端口以及外设地址都被映射于 I/O 空间内，所有的 I/O 地址都可以通过 IN 与 OUT 指令来访问，在 32 个通用工作寄存器和 I/O 地址之间传输数据。地址为 0x00 ~ 0x1F 的 I/O 寄存器还可使用 SBI 和 CBI 指令直接进行位寻址，而 SBIS 和 SBIC 指令则用来检查某一位的值。



#### 注意

使用 IN 和 OUT 指令时地址必须在 0x00 ~ 0x3F 之间，如果要像 SRAM 一样通过 LD 和 ST 指令访问 I/O 寄存器，则相应的地址要加上 0x20。

### 1.4.3 E<sup>2</sup>PROM 存储器

ATmega16 的 E<sup>2</sup>PROM 存储器常常用来存放一些需要掉电后保存的数据，其具体的使用方法参考第 10 章。



## 1.5 ATmega16 单片机的系统时钟

### 1.5.1 ATmega16 的系统时钟组成

ATmega16 的系统时钟由 CPU 时钟、I/O 时钟、FLASH 时钟、异步定时器时钟、ADC 时钟组成，在实际工作中使用这些时钟的不同选择和组合来驱动 ATmega16 工作，这些时钟的详细描述如下，结构如图 1.8 所示。

- CPU 时钟。 $\text{clk}_{\text{CPU}}$  与 ATmega16 的内核各个模块相连接，如通用寄存器、状态寄存器及保存堆栈指针的数据存储器，终止 CPU 时钟将使内核停止工作。
- I/O 时钟。 $\text{clk}_{\text{I/O}}$  用于驱动主要的 I/O 模块，如定时/计数器模块、SPI 总线接口模块和 USART 模块等，I/O 时钟还用于驱动外部中断模块。但是有些外部中断由异步逻辑检测，因此即使 I/O 时钟停止了这些中断仍然可以得到监控。另外 TWI (I<sup>2</sup>C) 总线接口模块的启动检测在没有  $\text{clk}_{\text{I/O}}$  的情况下也是使用异步逻辑来检测的，所以该数据总线模块在 ATmega16 的任何睡眠模式下都可以正常工作。
- FLASH 时钟。 $\text{clk}_{\text{FLASH}}$  用于控制 FLASH 存储器接口的操作，此时钟通常与 CPU 时钟同时挂起或激活。
- 异步定时器时钟。 $\text{clk}_{\text{ASY}}$  异步定时器时钟允许定时/计数器由外部 32kHz 时钟晶体驱动，使得这些定时/计数器即使在 ATmega16 的睡眠模式下仍然可以正常工作。
- ADC 时钟。 $\text{clk}_{\text{ADC}}$  是 ADC 模块的专用时钟，这样可以在 ADC 工作的时候停止 CPU 和 I/O 时钟以降低数字电路产生的噪声，从而提高 ADC 转换精度。

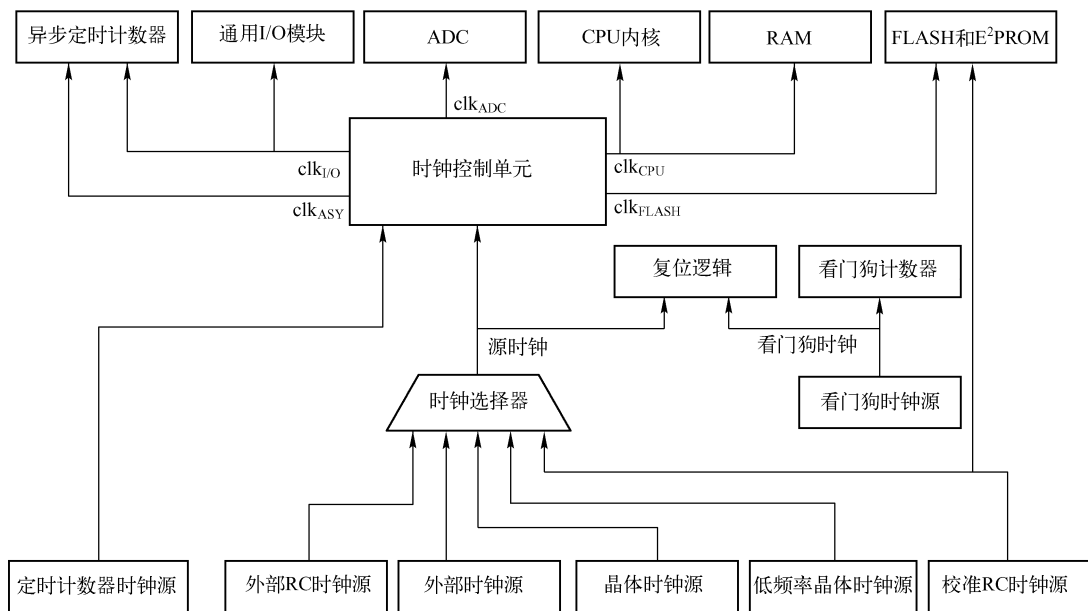


图 1.8 ATmega16 的系统时钟构成





### 1.5.2 ATmega16 的时钟源选择

ATmega16 可以通过对熔丝位的配置来选择时钟源，其对应的配置关系如表 1.3 所示。

表 1.3 使用熔丝位配置时钟源

时钟源选择	CKSEL3 ~ CKSEL0
外部晶体	1111 ~ 1010
外部低频晶振	1001
外部 RC 振荡器	1000 ~ 0101
标定的内部 RC 振荡器	0100 ~ 0001
外部时钟	0000

### 1.5.3 晶体振荡器

ATmega16 使用晶体振荡器的电路如图 1.9 所示，引脚 XTAL1 与 XTAL2 分别为用作片内振荡器的反向放大器的输入和输出，这个振荡器可以使用石英晶体，也可以使用陶瓷谐振器。

熔丝位 CKOPT 用来选择这两种放大器模式的其中之一，当 CKOPT 被编程时振荡器在输出引脚产生满幅度的振荡，这种模式适合于噪声环境，以及需要通过 XTAL2 驱动第二个时钟缓冲器的情况；而且这种模式的频率范围比较宽，当保持 CKOPT 为未编程状态时，振荡器的输出信号幅度比较小。其优点是大大降低了功耗，但是频率范围比较窄，而且不能驱动其他时钟缓冲器。

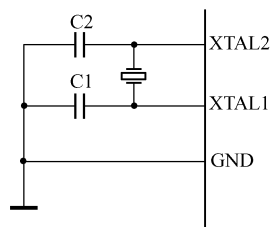


图 1.9 ATmega16 使用外部晶体振荡器作为时钟源

对于谐振器，CKOPT 未编程时的最大频率为 8MHz，CKOPT 编程时为 16MHz，电容 C1 和 C2 的数值要一样，不管使用的是晶体还是谐振器。最佳的数值与使用的晶体或谐振器有关，还与杂散电容和环境的电磁噪声有关，表 1.4 是针对晶体选择电容的一些常用参数。

振荡器可以工作于三种不同的模式，每一种都有一个优化的频率范围。工作模式通过熔丝位 CKSEL3 ~ CKSEL1 来选择，如表 1.4 所示。

表 1.4 晶体振荡器的相关参数选择

CKPOT	CKSEL3 ~ CKSEL0	频率 (MHz)	推荐电容值 (pF)
1	101	0.4 ~ 0.9	—
1	110	0.9 ~ 3.0	12 ~ 22
1	111	3.0 ~ 8.0	12 ~ 22
0	101、110、111	≥ 1.0	12 ~ 22

熔丝位 CKSEL0 和 SUT1 ~ SUT0 决定了 ATmega16 的启动时间，如表 1.5 所示。



表 1.5 ATmega16 的启动时间

CKSEL0	SUT1 ~ SUT0	启动时间	额外延长时间	推荐用法
0	00	258CK	4.1ms	陶瓷谐振器，电源快速上升
0	01	258CK	65ms	陶瓷谐振器，电源缓慢上升
0	10	1K CK	—	陶瓷谐振器，BOD 使能
0	11	1K CK	4.1ms	陶瓷谐振器，电源快速上升
1	00	1K CK	65ms	陶瓷谐振器，电源缓慢上升
1	01	16K CK	—	石英振荡器，BOD 使能
1	10	16K CK	4.1ms	石英振荡器，电源快速上升
1	11	16K CK	65ms	石英振荡器，电源缓慢上升

### 1.5.4 低频晶体振荡器

ATmega16 可以使用 32.768kHz 的钟表晶体作为器件的时钟源，此时必须将熔丝位 CKSEL 设置为“1001”以选择低频晶体振荡器，该晶体的连接方式和使用晶体振荡器相同，如图 1.9 所示。通过对熔丝位 CKOPT 的编程，用户可以使能 XTAL1 和 XTAL2 的内部电容，从而节省掉外部电容，内部电容的标称数值为 36pF，使用低频晶体振荡器之后，启动时间由熔丝位 SUT 确定，如表 1.6 所示。

表 1.6 低频晶体振荡器的启动时间

SUT1 ~ SUT0	启动时间	额外延长时间	推荐用法
00	1K CK	4.1ms	电源缓慢上升，BOD 使能
01	1K CK	65ms	电源缓慢上升
10	32K CK	65ms	启动时频率已经锁定
11	保留		

### 1.5.5 外部 RC 振荡器

如果 ATmega16 的应用系统不需要特别精确的定时，则可以使用如图 1.10 的外部 RC 振荡器作为时钟源，其频率可以通过公式

$$f = \frac{1}{3RC}$$

进行大略的计算，其中电容 C 至少要 22pF。通过对熔丝位 CKOPT 的编程，用户可以使能 XTAL1 和 GND 之间的片内 36pF 电容，从而无须外加电容。

RC 振荡器可以工作于四个不同的模式，每个模式有自己的优化频率范围，工作模式通过对熔丝位 CKSEL3 ~ CKSEL0 的编程来决定，如表 1.7 所示。

RC 振荡器的启动时间由熔丝位 SUT 决定，如表 1.8 所示。

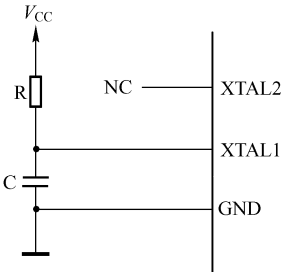


图 1.10 外部 RC 振荡器



表 1.7 RC 振荡器的工作模式

CKSEL3 ~ CKSEL0	频率 (MHz)
0101	$\leq 0.9$
0110	0.9 ~ 3.0
0111	3.0 ~ 8.0
1000	8.0 ~ 12.0

表 1.8 RC 振荡器启动时间

SUT1 ~ SUT0	启动时间	额外延长时间 ( $V_{CC} = 5.0V$ )	推荐用法
00	18CK	—	BOD 使能
01	18CK	4.1ms	电源快速上升
10	18CK	65ms	电源缓慢使能
11	6CK	4.1ms	电源快速上升或者 BOD 使能

### 1.5.6 片内 RC 振荡器

ATmega16 有一个经过校准的片内 RC 振荡器, 可以给 ATmega16 提供固定的 1.0、2.0、4.0 或 8.0MHz 的时钟 (这些频率都是 5V、25℃ 下的标称数值)。这个时钟使用熔丝位 CKSEL 进行控制, 如表 1.9 所示, 当使用这个时钟 (此时不能对 CKOPT 进行编程) 时无须外部振荡器件。当 ATmega16 复位时硬件将标定字节加载到 OSCCAL 寄存器中, 自动完成对内部 RC 振荡器的标定。在 5V, 25℃ 和频率为 1.0MHz 时, 这种标定可以提供标称频率  $\pm 3\%$  的始终精度, 在使用这个振荡器作为系统时钟时, 看门狗仍然使用自己的看门狗定时器作为溢出复位的依据。

表 1.9 使用校准片内 RC 时钟源

CKSEL3 ~ CKSEL0	频率 (MHz)
0001	1.0
0010	2.0
0011	4.0
0100	8.0



#### 注意

芯片出厂默认设置为 1.0MHz。

当使用 RC 时钟源时, ATmega16 的启动时间由熔丝位 SUT 决定, 如表 1.10 所示。

表 1.10 校准片内 RC 振荡器启动时间

SUT1 ~ SUT0	启动时间	额外延长时间 ( $V_{CC} = 5.0V$ )	推荐用法
00	6CK	—	BOD 使能
01	6CK	4.1ms	电源快速上升
10	6CK	65ms	电源缓慢使能
11	保留		





表 1.11 是振荡器标定寄存器 OSCCAL 的内部结构。将标定数据写入这个地址可以对内部振荡器进行调节以消除由于生产工艺所带来的振荡器频率偏差。复位时 1MHz 的标定数据（标识数据的高字节，地址为 0x00）自动加载到 OSCCAL 寄存器。

表 1.11 OSCCAL 寄存器

BIT	CAL7	CAL6	CAL5	CAL4	CAL3	CAL2	CAL1	CAL0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	标定数据							

如果需要内部 RC 振荡器工作于其他频率，其标定数据必须人工加载：首先通过编程器读取标识数据，然后将标定数据保存到 FLASH 或 E<sup>2</sup>PROM 之中。这些数据可以通过软件读取，然后加载到 OSCCAL 寄存器。当 OSCCAL 为零时，振荡器以最低频率工作，当对其写如不为零的数据时，内部振荡器的频率将增长，写入 0xFF 即得到最高频率，标定的振荡器将用来为访问 E<sup>2</sup>PROM 和 FLASH 定时。需要注意的是，有写 E<sup>2</sup>PROM 和 FLASH 的操作时不要将频率标定到超过标称频率的 10%，否则写操作有可能失败，如表 1.12 所示。



#### 注意

振荡器只对 1.0、2.0、4.0 和 8.0MHz 这 4 种频率进行了标定，其他频率则无法保证。

表 1.12 内部校准 RC 时钟源范围

OSCAL 数值	最小频率 (%)	最大频率 (%)
0x00	50	100
0x7F	75	150
0xFF	100	200

### 1.5.7 外部时钟源

ATmega16 可以使用外部时钟源作为芯片驱动，此时 XTAL1 引脚必须如图 1.11 所示进行连接，同时熔丝位 CKSEL 必须编程为“0000”，若熔丝位 CKOPT 也被编程，用户就可以使用内部的 XTAL1 和 GND 之间的 36pF 电容，此时不需要外加电容。

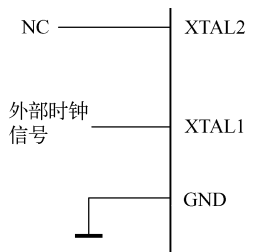


图 1.11 外部时钟源电路图

当选择了外部时钟源之后，ATmega16 的启动时间由熔丝位 SUT 确定，如表 1.13 所示。为了保证 ATmega16 能够稳定工作，不能突然改变外部时钟源的振荡频率，当工作频率突变超过 2% 将会产生异常现象，所以应该在 ATmega16 保持复位状态时改变外部时钟的振荡频率。晶体可以直接与这两个引脚连接，无须外部电容，此振荡器针对 32.768kHz 的钟表晶体做了优化，不建议在 TOSC1 引脚输入振荡信号。



表 1.13 外部时钟源启动时间

SUT1 ~ SUT0	启动时间	额外延长时间 ( $V_{CC} = 5.0V$ )	推荐用法
00	6CK	—	BOD 使能
01	6CK	4.1ms	电源快速上升
10	6CK	65ms	电源缓慢使能
11	保留		

## 1.6 ATmega16 单片机的电源管理

ATmega16 有多种电源管理模式, 通过对控制寄存器 MCUCR 的设置可以控制 ATmega16 进入不同的电源管理模式, MCUCR 寄存器的内部结构如表 1.14 所示, 其中只有 4 位和 ATmega16 的电源管理相关。

表 1.14 ATmega16 的控制寄存器 MCUCR

BIT	SRE	SRW10	SE	SM1	SM0	SM2	IVSEL	IVCE
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- SE。睡眠使能控制位, 当 SE 被置位时 ATmega16 进入睡眠模式。
- SM2 ~ SM0。电源模式选择位, 用于控制 ATmega16 的具体电源模式, 如表 1.15 所示。

表 1.15 ATmega16 电源模式控制选择

SM2	SM1	SM0	休眠模式
0	0	0	空闲模式
0	0	1	ADC 噪声抑制模式
0	1	0	掉电模式
0	1	1	省电模式
1	0	0	保留
1	0	1	保留
1	1	0	StandBy 模式
1	1	1	扩展 StandBy 模式



### 注意

只有在使用外部晶体或谐振器时, StandBy 模式与扩展 StandBy 模式才可用。

- 空闲模式。当 SM2 ~ SM0 = “000” 时, SLEEP 指令使 ATmega16 进入空闲模式。在此模式下, CPU 停止运行, 而 SPI、USART、模拟比较器、ADC、两线串行接口、定时/计数器、看门狗和中断系统继续工作。此模式只停止 CPU 时钟和 FLASH 时钟, 其他时钟则继续工作。类似定时器溢出与 USART 传输完成等内外部中断都可以唤醒 ATmega16, 如果不需要从模拟比较器中断唤醒, 为了减少功耗, 可以切断比较器的电源, 方法是置位模拟比较器控制和状态寄存器 ACSR 的 ACD 位, 如果 ADC 通道被



使能，进入此模式后将自动启动一次转换。

- **ADC 噪声抑制模式。**当  $SM2 \sim SM0 = "001"$  时，SLEEP 指令使 ATmega16 进入噪声抑制模式。在此模式下，CPU 停止运行，而 ADC、外部中断、两线串行接口、定时/计数器 2 和看门狗继续工作。此模式只停止了 I/O 时钟、CPU 时钟和 FLASH 时钟，其他时钟则继续工作。此模式改善了 ADC 的噪声环境，使得转换精度更高，当 ADC 被使能的时候，进入此模式将自动启动一次 ADC 转换，此后每次 ADC 的转换结束中断、外部复位、看门狗复位、BOD 复位、两线串行接口地址匹配中断、定时/计数器 0 中断、SPM/E<sup>2</sup>PROM 准备就绪中断、外部电平中断 INT7  $\sim$  INT4 或者外部中断 INT3  $\sim$  INT0 可以将 MCU 从 ADC 噪声抑制模式唤醒。
- **掉电模式。**当  $SM2 \sim SM0 = "010"$  时，SLEEP 指令使 ATmega16 进入掉电模式。在此模式下，外部晶体停振，而外部中断、两线串行接口、看门狗（如果使能的话）继续工作。只有外部复位、看门狗复位、BOD 复位、两线串行接口地址匹配中断、外部电平中断 INT7  $\sim$  INT4 或者外部中断 INT3  $\sim$  INT0 可以使 ATmega16 脱离掉电模式，此模式下 ATmega16 停止了所有的时钟，只有异步模块可以继续工作。



#### 注意

当使用外部电平中断方式将 ATmega16 从掉电模式唤醒时，必须保持外部电平一定的时间，并且从施加掉电唤醒条件到真正唤醒有一个延迟时间，此时间用于时钟重新启动并稳定下来，唤醒周期与由熔丝位 CKSEL 定义的复位周期是一样的。

- **省电模式。**当  $SM2 \sim SM0 = "011"$  时，SLEEP 指令将使 ATmega16 进入省电模式。该模式与掉电模式只有一点不同：如果定时计数器 2 为异步驱动，则定时计数器 2 在睡眠时将继续运行。除了掉电模式的唤醒方式，定时计数器 2 的溢出中断和比较匹配中断也可以将 ATmega16 从休眠方式唤醒，只要 TIMSK 使能了这些中断，且 SREG 的全局中断使能位 I 被置位，此模式停止了除 ASY 时钟以外所有的时钟，只有异步模块可以继续工作。



#### 说明

如果异步定时器不是异步驱动的，建议使用掉电模式，而不是省电模式。

- **StandBy 模式。**当  $SM2 \sim SM0 = "110"$  时，SLEEP 指令使 ATmega16 进入 StandBy 模式，该模式与掉电模式唯一的不同之处在于振荡器继续工作。其唤醒时间只需要 6 个时钟周期。
- **扩展 StandBy 模式。**当  $SM2 \sim SM0 = "111"$  时，SLEEP 指令将使 ATmega16 进入扩展的 StandBy 模式，该模式与省电模式、掉电模式唯一的不同之处在于振荡器继续工作，其唤醒时间只需要 6 个时钟周期。

为了减少 ATmega16 应用系统的功耗，所以需要考虑几个问题，一般来说，要尽可能利用睡眠模式并且使尽可能少的模块继续工作，不需要的功能必须禁止。下面的模块需要特殊考虑以达到尽可能低的功耗。

- **模数转换器。**被使能时，ADC 在睡眠模式下继续工作，为了降低功耗，在进入睡眠





模式之前需要禁止 ADC 通道。

- 模拟比较器。在空闲模式时，如果没有使用模拟比较器，可以将其关闭，在 ADC 噪声抑制模式下也是如此。在其他模式下模拟比较器是自动关闭的，如果模拟比较器使用了内部电压基准源，则不论在什么模式下都需要关闭它，否则内部电压基准源将一直被使能。
- 掉电检测 BOD。如果系统没有使用掉电检测器 BOD，那么这个模块也可以关闭，如果熔丝位 BODEN 被编程，从而使能了 BOD 功能，则其将在各种模式下继续工作，这个电流将占总电流的很大比重。
- 片内基准电压。当使用 BOD、模拟比较器和 ADC 时可能需要内部电压基准源，若这些模块都被禁止了，则基准源也可以被禁止，但是重新使能后用户必须等待基准源稳定之后才可以使它。如果基准源在休眠过程中是使能的，其输出则立即可以使用。
- 看门狗定时器。如果系统无须使用看门狗，这个模块也可以关闭，若使能，则在任何模式下都将持续工作从而消耗电流。
- I/O 端口引脚。进入休眠模式时，所有的端口引脚都应该配置为只消耗最小的功耗。最重要的是避免驱动电阻性负载。在休眠模式下 I/O 时钟和 ADC 时钟都被停止，输入缓冲器也被禁止，从而保证输入电路不会消耗电流。而在某些情况下输入逻辑是使能的，用来检测唤醒条件。如果输入缓冲器是使能的，此时输入不能悬空，信号电平也不应该接近  $V_{CC}/2$ ，否则输入缓冲器会消耗额外的电流。
- JTAG 接口与片上调试系统。如果通过熔丝位 OCDEN 使能了片上调试系统，当 ATmega16 进入掉电或省电模式时主时钟将保持运行，可以使用不编程 OCDEN、不编程 JTAGEN、置位 MCUCSR 寄存器的 JTD 位的方法来减少功耗。当 JTAG 接口被使能而 JTAGTAP 控制器没有进行数据交换时，引脚 TDO 将悬空，如果与 TDO 引脚连接的硬件电路没有上拉电阻，系统功耗将增加，而 ATmega16 的引脚 TDI 包含一个上拉电阻，因此在扫描链中无须为下一个芯片的 TDO 引脚设置上拉电阻。通过置位 MCUCSR 寄存器的 JTD 或不对 JTAG 熔丝位编程可以禁止 JTAG 接口。

## 1.7 ATmega16 单片机的复位

当 ATmega16 复位之后，所有的 I/O 寄存器都被恢复为初始值，程序从复位向量处开始执行，复位向量处存放的指令必须是绝对跳转指令 JMP，以使程序跳转到对应复位处理代码。如果用户代码不使用中断，中断向量的存储位置可以由一般的程序代码所覆盖，这个处理方法同样适用于当复位向量位于应用程序区，中断向量位于程序存储器的 Boot 区的状况。

当复位发生时，ATmega16 的 I/O 端口立即被复位为初始值，此时不要求任何时钟处于正常运行状态，当所有的复位信号消失之后，ATmega16 内部的一个延迟计数器被激活，将内部复位的时间延长，从而能使得在 ATmega16 在正常工作之前有一定的时间让电源达到稳定的电平，该延迟计数器的溢出时间通过熔丝位 SUT 与 CKSEL 设定，参考 1.5.3 小节。

### 1.7.1 ATmega16 的复位源

ATmega16 有如下 5 个复位源。

- (1) 上电复位。当电源电压低于上电复位门限  $V_{POT}$  时，ATmega16 复位。
- (2) 外部复位。当外加在 RESET 引脚上的低电平持续时间大于最小脉冲宽度时，ATmega16 复位。
- (3) 掉电检测复位。当掉电检测复位功能使能，并且电源电压低于掉电检测复位门限  $V_{BOT}$  时，ATmega16 被复位。
- (4) 看门狗复位。当看门狗使能，并且看门狗定时器溢出时，ATmega16 复位。
- (5) JTAG 复位。当复位寄存器被置位时，ATmega16 复位。

### 1.7.2 上电复位

在上电复位（POR）过程中，其复位脉冲由片内检测电路产生，当 VCC 引脚上的电平低于检测电平 POR 时，发生上电复位事件，POR 电路可以用来触发启动复位，或者用来检测电源故障。

POR 电路可以保证 ATmega16 在上电时复位，当  $V_{CC}$  达到上电门限电压后触发延迟计数器，在计数器溢出之前器件一直保持为复位状态，当  $V_{CC}$  下降时，只要低于检测门限，RESET 信号立即生效。

图 1.12 和图 1.13 是 ATmega16 在上电复位时 RESET 引脚不同连接方式下的信号时序图。

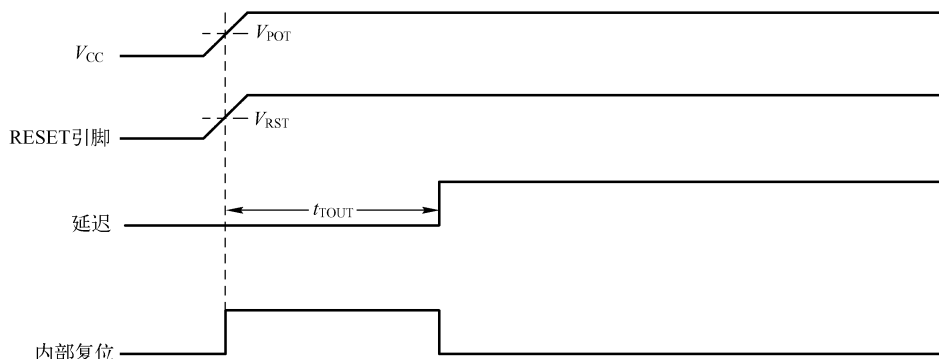


图 1.12 RESET 引脚连接到 VCC 引脚

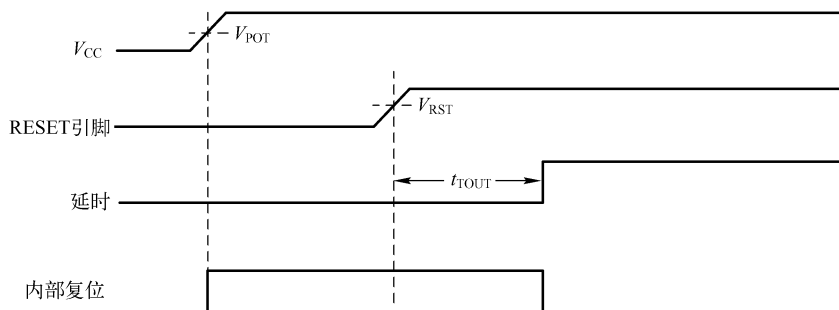


图 1.13 RESET 由外部电路控制





### 1.7.3 外部复位

外部复位由外加于 ATmega16 的 RESET 引脚上的低电平产生, 当复位低电平持续时间大于最小脉冲宽度 ( $1.5\mu\text{s}$ ) 时即触发复位过程, 即使此时并没有时钟信号在运行。当外加信号达到复位门限电压  $V_{\text{RST}}$  (上升沿) 时,  $t_{\text{TOUT}}$  延时周期开始, 在延时结束后 ATmega16 启动, 如图 1.14 所示。

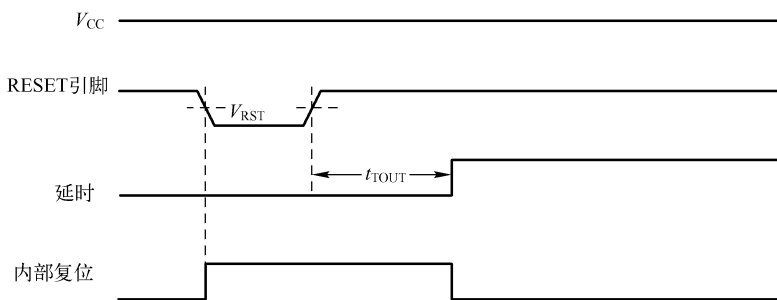


图 1.14 外部复位

### 1.7.4 掉电检测复位

ATmega16 具有片内 BOD (Brown-out Detect/On) 电路, 该电路通过与固定的触发电平的对比来检测工作过程中  $V_{\text{CC}}$  的变化, 此触发电平通过熔丝位 BODLEVEL 来设定, 可以设定为 2.7V 或 4.0V。BOD 的触发电平具有迟滞功能以消除电源尖峰的影响, 其电路的开关由熔丝位 BODEN 来控制, 当 BOD 被使能后一旦  $V_{\text{CC}}$  下降到触发电平以下, BOD 复位立即被激发, 当  $V_{\text{CC}}$  上升到触发电平以上时延时计数器开始计数, 一旦超过溢出时间  $t_{\text{TOUT}}$ , ATmega16 即恢复工作。如果  $V_{\text{CC}}$  一直低于触发电平并保持时间  $t_{\text{BOD}}$  ( $2\mu\text{s}$ ), BOD 电路将只检测电压跌落, 图 1.15 是掉电检测复位的示意图。

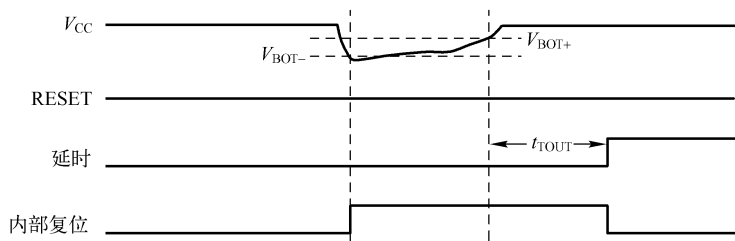


图 1.15 掉电检测复位

### 1.7.5 看门狗复位

当看门狗定时器溢出时将产生持续时间为 1 个 CK 周期的复位脉冲, 在脉冲的下降沿, 延时定时器开始对  $t_{\text{TOUT}}$  计数, 计数完成之后 ATmega16 即完成复位, 如图 1.16 所示。

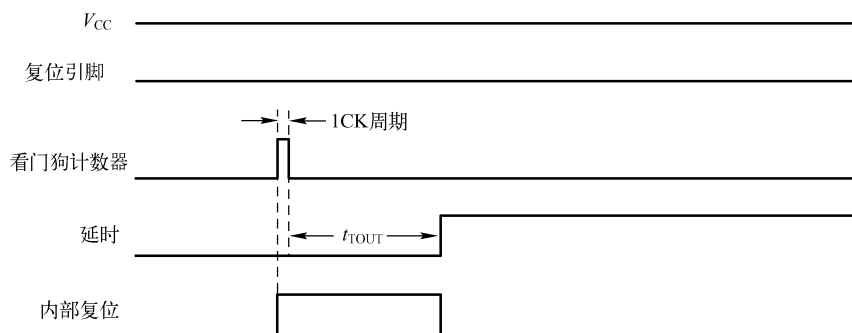


图 1.16 看门狗复位

### 1.7.6 ATmega16 的复位控制寄存器

ATmega16 使用一个复位控制寄存器来对复位进行管理，该寄存器的内部结构如表 1.16 所示，其中后 5 位和 ATmega16 的复位控制相关，其详细说明如下。

表 1.16 ATmega16 的复位控制寄存器 MCUCSR

BIT	JTD	ISC2	—	JTRF	WDRF	BORF	EXTRF	PORF
读/写	R/W	R	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	具体情况				

- JTRF: JTAG 复位标志位。通过 JTAG 指令 AVR\_RESET 可以使 JTAG 复位寄存器置位，引发 ATmega16 复位，并使 JTRF 置位，上电复位后该位被自动清零，也可以通过对该位写“0”来清除。
- WDRF: 看门狗复位标志位。当看门狗复位发生时该位被置位，上电复位后该位被自动清零，也可以通过向该位写“0”来清除。
- BORF: 掉电检测复位标志位。在掉电检测复位发生时被置位，上电复位后该位被自动清零，也可以通过写“0”来清除。
- EXTRF: 外部复位标志位。当外部复位发生时被置位，上电复位后该位被自动清零，也可以通过写“0”来清除。
- PORF: 上电复位标志位。上电复位发生时被置位，只能通过写“0”来清除。

### 1.7.7 片内基准电压

ATmega16 具有片内基准电压源，用于掉电检测，或者作为模拟比较器或 ADC 模块的输入。ADC 模块的 2.56V 基准电压也由此片内基准电压源产生。电压基准的启动时间可能影响其工作方式，其启动时间如表 1.17 所示。为了降低功耗，在不使用基准源时关闭这个基准电压源，该基准源仅在如下三种情况时打开。

- BOD 使能，即熔丝位 BODEN 被编程。
- 能隙基准源连接到模拟比较器，ACSR 寄存器的 ACBG 位被置位。
- ADC 模块被使能。



因此，当 BOD 被禁止时，置位 ACBG 或使能 ADC 后会启动基准源，为了降低掉电模式的功耗，可以禁止上述三种条件，并在进入掉电模式之前关闭基准源。

表 1.17 内部电压基准源特性

符 号	参 数	最 小 值	典 型 值	最 大 值	单 位
$V_{BG}$	能隙基准源电压	1.15	1.23	1.35	V
$t_{BG}$	能隙基准源启动时间		40	70	$\mu s$
$I_{BG}$	能隙基准源功耗		10		$\mu A$

## 1.8 ATmega16 单片机的中断系统

表 1.18 是 ATmega16 的中断向量列表，ATmega16 共有 21 个中断向量。

表 1.18 ATmega16 的中断向量列表

向量号	程序地址	中 断 源	说 明
1	0x0000	复位	外部引脚电平引发的复位，上电复位，掉电检测复位，看门狗复位，以及 JTAG AVR 复位
2	0x0002	INT0	外部中断 0
3	0x0004	INT1	外部中断 1
4	0x0006	TIMER2 COMP	定时计数器 2 比较匹配
5	0x0008	TIMER2 OVR	定时计数器 2 溢出
6	0x000A	TIMER1 CAPT	定时计数器 1 事件捕捉
7	0x000C	TIMER1 COMPA	定时计数器 1 比较匹配 A
8	0x000E	TIMER1 COMPB	定时计数器 1 比较匹配 B
9	0x0010	TIMER1 OVF	定时计数器 1 溢出
10	0x0012	TIMER0 OVF	定时计数器 0 溢出
11	0x0014	SPI, STC	SPI 串行传输结束
12	0x0016	USART, RXC	USART，接收中断
13	0x0018	USART, UDRE	USART，数据寄存器空
14	0x001A	USART, TXC	USART，发送中断
15	0x001C	ADC	ADC 转换结束
16	0x001E	EE_RDY	E <sup>2</sup> PROM 空闲
17	0x0020	ANA_COMP	模拟比较器比较完成
18	0x0022	TWI	两线串行接口模块中断
19	0x0024	INT2	外部中断请求 2
20	0x0026	TIMER0 COMP	定时器/计数器 0 比较匹配
21	0x0028	SPM READY	保存程序存储器内容就绪



### 说明

当熔丝位 BOOTRST 被编程时，ATmega16 复位后程序将跳转到 Boot Loader。当寄存器 MCUCR 的 IVSEL 位被置位时，中断向量将转移到 Boot 区的起始地址，此时各个中断向量的实际地址为表中地址与 Boot 区起始地址之和。





表 1.19 给出了不同的 BOOTRST、IVSEL 位设置下的复位和中断向量的位置，如果程序永远不使能中断，中断向量就没有意义，用户则可以在此直接写程序。同样，如果复位向量位于应用区，而其他中断向量位于 Boot 区，则复位向量之后可以直接写程序，反之也是如此。

表 1.19 复位和中断向量的位置

BOOTRST	IVSEL	复 位 地 址	中断向量起始地址
1	0	0x0000	0x0002
1	1	0x0000	Boot 区复位地址 + 0x0002
0	0	Boot 区复位地址	0x0002
0	1	Boot 区复位地址	Boot 区复位地址 + 0x0002

可以使用通用中断控制寄存器 GICR 来控制 ATmega16 的相关中断，表 1.20 是 GICR 寄存器的内部结构，其中 IVSEL 和 IVCE 位用于对 ATmega16 的通用中断进行控制。

表 1.20 ATmega16 的通用中断控制寄存器 GICR

BIT	INT1	INT0	INT2	—	—	—	IVSEL	IVCE
读/写	R/W	R/W	R/W	R	R	R	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- IVSEL。中断向量选择位，当 IVSEL 为“0”时，中断向量位于 FLASH 存储器的起始地址；当 IVSEL 为“1”时，中断向量转移到 Boot 区的起始地址。实际的 Boot 区起始地址由熔丝位 BOOTSZ 确定。
- IVCE。中断向量修改使能位，在修改 IVSEL 时 IVCE 必须置位，在 IVCE 或 IVSEL 写操作 4 个时钟周期后，IVCE 位被硬件清零。

为了防止无意识地改变中断向量表，修改 IVSEL 位时需要遵照如下过程。

(1) 置位中断向量修改使能位 IVCE。

(2) 在接下来的 4 个时钟周期里将修改 IVSEL 位，同时对 IVCE 写“0”。

(3) 执行上述序列时中断自动被禁止。在置位 IVCE 时中断就被禁止了，并一直保持到写 IVSEL 操作之后的下一条语句，如果没有 IVSEL 写操作，则中断在置位 IVCE 之后的 4 个时钟周期保持禁止。需要注意的是，虽然中断被自动禁止，但状态寄存器的位 I 的值并不受此操作的影响。



#### 说明

若中断向量位于 Boot 区，且 Boot 锁定位 BLB02 被编程，则执行应用区的程序时中断被禁止；若中断向量位于应用区，且 Boot 锁定位 BLB12 被编程，则执行 Boot 区的程序时中断被禁止。

## 第2章 ATmega16 单片机的指令和 C 语言

指令是 ATmega16 单片机能执行的命令，其集合即为指令系统，属于 AVR 系列单片机 RISC 结构的精简指令集的一部分，本章详细介绍这些指令的功能，但是使用指令系统进行程序编写效率较低，所以本章还会介绍如何使用 C 语言进行 ATmega16 单片机的开发。

### 2.1 ATmega16 单片机的指令系统

ATmega16 单片机的指令系统可以分为指令集和寻址方式两部分。

#### 2.1.1 ATmega16 单片机的指令集

ATmega16 单片机的指令集可以分为算术运算和逻辑运算指令、跳转指令、数据传送指令、位和位测试指令、控制指令等五个大类，其详细说明如表 2.1 所示。

表 2.1 ATmega16 的指令集

指令	操作数	说 明	操 作	标 志	时钟数
算术和逻辑指令					
ADD	Rd, Rr	无进位加法	$Rd \leftarrow Rd + Rr$	Z, C, N, V, H	1
ADC	Rd, Rr	带进位加法	$Rd \leftarrow Rd + Rr + C$	Z, C, N, V, H	1
ADIW	Rdl, K	立即数与字相加	$Rdh; Rdl \leftarrow Rdh; Rdl + K$	Z, C, N, V, S	2
SUB	Rd, Rr	无进位减法	$Rd \leftarrow Rd - Rr$	Z, C, N, V, H	1
SUBI	Rd, K	减立即数	$Rd \leftarrow Rd - K$	Z, C, N, V, H	1
SBC	Rd, Rr	带进位减法	$Rd \leftarrow Rd - Rr - C$	Z, C, N, V, H	1
SBCI	Rd, K	带进位减立即数	$Rd \leftarrow Rd - K - C$	Z, C, N, V, H	1
SBIW	Rdl, K	从字中减立即数	$Rdh; Rdl \leftarrow Rdh; Rdl - K$	Z, C, N, V, S	2
AND	Rd, Rr	逻辑与	$Rd \leftarrow Rd \cdot Rr$	Z, N, V	1
ANDI	Rd, K	与立即数的逻辑与	$Rd \leftarrow Rd \cdot K$	Z, N, V	1
OR	Rd, Rr	逻辑或	$Rd \leftarrow Rd \vee Rr$	Z, N, V	1
ORI	Rd, K	与立即数的逻辑或	$Rd \leftarrow Rd \vee K$	Z, N, V	1
EOR	Rd, Rr	异或	$Rd \leftarrow Rd \oplus Rr$	Z, N, V	1
COM	Rd	1 的补码	$Rd \leftarrow 0xFF - Rd$	Z, C, N, V	1
NEG	Rd	2 的补码	$Rd \leftarrow 0x00 - Rd$	Z, C, N, V, H	1
SBR	Rd, K	设置寄存器的位	$Rd \leftarrow Rd \vee K$	Z, N, V	1
CBR	Rd, K	寄存器位清零	$Rd \leftarrow Rd \cdot (0xFF - K)$	Z, N, V	1
INC	Rd	加 1 操作	$Rd \leftarrow Rd + 1$	Z, N, V	1
DEC	Rd	减 1 操作	$Rd \leftarrow Rd - 1$	Z, N, V	1

续表

指令	操作数	说 明	操 作	标 志	时钟数
算术和逻辑指令					
TST	Rd	测试是否为零或负	$Rd \leftarrow Rd \cdot Rd$	Z, N, V	1
CLR	Rd	寄存器清零	$Rd \leftarrow Rd \oplus Rd$	Z, N, V	1
SER	Rd	寄存器置位	$Rd \leftarrow 0xFF$	无	1
MUL	Rd, Rr	无符号数乘法	$R1; R0 \leftarrow Rd \times Rr$	Z, C	2
MULS	Rd, Rr	有符号数乘法	$R1; R0 \leftarrow Rd \times Rr$	Z, C	2
MULSU	Rd, Rr	有符号数与无符号数乘法	$R1; R0 \leftarrow Rd \times Rr$	Z, C	2
FMUL	Rd, Rr	无符号小数乘法	$R1; R0 \leftarrow (Rd \times Rr) \ll 1$	Z, C	2
FMULS	Rd, Rr	有符号小数乘法	$R1; R0 \leftarrow (Rd \times Rr) \ll 1$	Z, C	2
FMULSU	Rd, Rr	有符号小数与无符号小数乘法	$R1; R0 \leftarrow (Rd \times Rr) \ll 1$	Z, C	2
跳 转 指 令					
RJMP	k	相对跳转	$PC \leftarrow PC + k + 1$	无	2
IJMP		间接跳转到(Z)	$PC \leftarrow Z$	无	2
JMP	k	直接跳转	$PC \leftarrow k$	无	3
RCALL	k	相对子程序调用	$PC \leftarrow PC + k + 1$	无	3
ICALL		间接调用(Z)	$PC \leftarrow Z$	无	3
CALL	k	直接子程序调用	$PC \leftarrow k$	无	4
RET		子程序返回	$PC \leftarrow Stack$	无	4
RETI		中断返回	$PC \leftarrow Stack$	I	4
CPSE	Rd, Rr	比较, 相等则跳过下一条指令	$\text{if}(Rd = Rr) PC \leftarrow PC + 2 \text{ or } 3$	无	1/2/3
CP	Rd, Rr	比较	$Rd - Rr$	Z, C, N, V, H	1
CPC	Rd, Rr	带进位比较	$Rd - Rr - C$	Z, C, N, V, H	1
CPI	Rd, K	与立即数比较	$Rd - K$	Z, C, N, V, H	1
SBRC	Rr, b	寄存器位为 0 则跳过下一条指令	$\text{if}(Rr(b) = 0) PC \leftarrow PC + 2 \text{ or } 3$	无	1/2/3
SBRSC	Rr, b	寄存器位为 1 则跳过下一条指令	$\text{if}(Rr(b) = 1) PC \leftarrow PC + 2 \text{ or } 3$	无	1/2/3
SBIC	P, b	I/O 寄存器位为 0 则跳过下一条指令	$\text{if}(P(b) = 0) PC \leftarrow PC + 2 \text{ or } 3$	无	1/2/3
SBIS	P, b	I/O 寄存器位为 1 则跳过下一条指令	$\text{if}(P(b) = 1) PC \leftarrow PC + 2 \text{ or } 3$	无	1/2/3
BRBS	s, k	状态寄存器位为 1 则跳过下一条指令	$\text{if}(SREG(s) = 1) \text{ then } PC \leftarrow PC + k + 1$	无	1/2/3
BRBC	s, k	状态寄存器位为 0 则跳过下一条指令	$\text{if}(SREG(s) = 0) \text{ then } PC \leftarrow PC + k + 1$	无	1/2
BREQ	k	相等则跳转	$\text{if}(Z = 1) \text{ then } PC \leftarrow PC + k + 1$	无	1/2
BRNE	k	不相等则跳转	$\text{if}(Z = 0) \text{ then } PC \leftarrow PC + k + 1$	无	1/2
BRCS	k	进位位为 1 则跳转	$\text{if}(C = 1) \text{ then } PC \leftarrow PC + k + 1$	无	1/2
BRCC	k	进位位为 0 则跳转	$\text{if}(C = 0) \text{ then } PC \leftarrow PC + k + 1$	无	1/2
BRSH	k	大于或等于则跳转	$\text{if}(C = 0) \text{ then } PC \leftarrow PC + k + 1$	无	1/2

续表

指令	操作数	说 明	操 作	标 志	时钟数
跳 转 指 令					
BRLO	k	小于则跳转	if( C = 1 ) then PC ← PC + k + 1	无	1/2
BRMI	k	负则跳转	if( N = 1 ) then PC ← PC + k + 1	无	1/2
BRPL	k	正则跳转	if( N = 0 ) then PC ← PC + k + 1	无	1/2
BRGE	k	有符号数大于或等于则跳转	if( N ⊕ V = 0 ) then PC ← PC + k + 1	无	1/2
BRLT	k	有符号数负则跳转	if( N ⊕ V = 1 ) then PC ← PC + k + 1	无	1/2
BRHS	k	半进位为 1 则跳转	if( H = 1 ) then PC ← PC + k + 1	无	1/2
BRHC	k	半进位为 0 则跳转	if( H = 0 ) then PC ← PC + k + 1	无	1/2
BRTS	k	T 为 1 则跳转	if( T = 1 ) then PC ← PC + k + 1	无	1/2
BRTC	k	T 为 0 则跳转	if( T = 0 ) then PC ← PC + k + 1	无	1/2
BRVS	k	溢出标志为 1 则跳转	if( V = 1 ) then PC ← PC + k + 1	无	1/2
BRVC	k	溢出标志为 0 则跳转	if( V = 0 ) then PC ← PC + k + 1	无	1/2
BRIE	k	中断使能再跳转	if( I = 1 ) then PC ← PC + k + 1	无	1/2
BRID	k	中断禁用再跳转	if( I = 0 ) then PC ← PC + k + 1	无	1/2
数据传送指令					
MOV	Rd, Rr	寄存器间复制	Rd ← Rr	无	1
MOVW	Rd, Rr	复制寄存器字	Rd + 1; Rd ← Rr + 1; Rr	无	1
LDI	Rd, K	加载立即数	Rd ← K	无	1
LD	Rd, X	加载间接寻址数据	Rd ← ( X )	无	1
LD	Rd, X +	加载间接寻址数据, 然后地址加 1	Rd ← ( X ), X ← X + 1	无	2
LD	Rd, - X	地址减 1 后加载间接寻址数据	X ← X - 1, Rd ← ( X )	无	2
LD	Rd, Y	加载间接寻址数据	Rd ← ( Y )	无	2
LD	Rd, Y +	加载间接寻址数据, 然后地址加 1	Rd ← ( Y ), Y ← Y + 1	无	2
LD	Rd, - Y	地址减 1 后加载间接寻址数据	Y ← Y - 1, Rd ← ( Y )	无	2
LDD	Rd, Y + q	加载带偏移量的间接寻址数据	Rd ← ( Y + q )	无	2
LD	Rd, Z	加载间接寻址数据	Rd ← ( Z )	无	2
LD	Rd, Z +	加载间接寻址数据, 然后地址加 1	Rd ← ( Z ), Z ← Z + 1	无	2
LD	Rd, - Z	地址减 1 后加载间接寻址数据	Z ← Z - 1, Rd ← ( Z )	无	2
LDD	Rd, Z + q	加载带偏移量的间接寻址数据	Rd ← ( Z + q )	无	2
LDS	Rd, k	从 SRAM 加载数据	Rd ← ( k )	无	2
ST	X, Rr	以间接寻址方式存储数据	( X ) ← Rr	无	2
ST	X +, Rr	以间接寻址方式存储数据, 然后地址加 1	( X ) ← Rr, X ← X + 1	无	2
ST	- X, Rr	地址减 1 后以间接寻址方式存储数据	X ← X - 1, ( X ) ← Rr	无	2
ST	Y, Rr	加载间接寻址数据	( Y ) ← Rr	无	2
ST	Y +, Rr	加载间接寻址数据, 然后地址加 1	( Y ) ← Rr, Y ← Y + 1	无	2
ST	- Y, Rr	地址减 1 后加载间接寻址数据	Y ← Y - 1, ( Y ) ← Rr	无	2
STD	Y + q, Rr	加载带偏移量的间接寻址数据	( Y + q ) ← Rr	无	2

续表

指令	操作数	说 明	操 作	标 志	时钟数
数据传送指令					
ST	Z, Rr	加载间接寻址数据	$(Z) \leftarrow Rr$	无	2
ST	Z + , Rr	加载间接寻址数据, 然后地址加 1	$(Z) \leftarrow Rr, Z \leftarrow Z +$	无	2
ST	- Z, Rr	地址减 1 后加载间接寻址数据	$Z \leftarrow Z - 1, (Z) \leftarrow Rr$	无	2
STD	Z + q, Rr	加载带偏移量的间接寻址数据	$(Z + q) \leftarrow Rr$	无	2
STS	k, Rr	从 SRAM 加载数据	$(k) \leftarrow Rr$	无	2
LPM		加载程序空间的数据	$R0 \leftarrow (Z)$	无	3
LPM	Rd, Z	加载程序空间的数据	$Rd \leftarrow (Z)$	无	3
LPM	Rd, Z +	加载程序空间的数据, 然后地址加 1	$Rd \leftarrow (Z), Z \leftarrow Z + 1$	无	3
SPM		保存程序空间的数据	$(Z) \leftarrow R1; R0$	无	
IN	Rd, P	从 I/O 端口读数据	$Rd \leftarrow P$	无	1
OUT	P, Rr	输出端口	$P \leftarrow Rr$	无	1
PUSH	Rr	将寄存器推入堆栈	$Stack \leftarrow Rr$	无	2
POP	Rd	将寄存器从堆栈中弹出	$Rd \leftarrow Stack$	无	2
位和位测试指令					
SBI	P, b	I/O 寄存器位置位	$I/O(P, b) \leftarrow 1$	无	2
CBI	P, b	I/O 寄存器位清零	$I/O(P, b) \leftarrow 0$	无	2
LSL	Rd	逻辑左移	$Rd(n+1) \leftarrow Rd(n), Rd(0) \leftarrow 0$	Z, C, N, V	1
LSR	Rd	逻辑右移	$Rd(n) \leftarrow Rd(n+1), Rd(7) \leftarrow 0$	Z, C, N, V	1
ROL	Rd	带进位循环左移	$Rd(0) \leftarrow C, Rd(n+1) \leftarrow Rd(n), C \leftarrow Rd(7)$	Z, C, N, V	1
ROR	Rd	带进位循环右移	$Rd(7) \leftarrow C, Rd(n) \leftarrow Rd(n+1), C \leftarrow Rd(0)$	Z, C, N, V	1
ASR	Rd	算术右移	$Rd(n) \leftarrow Rd(n+1), n=0 \sim 6$	Z, C, N, V	1
SWAP	Rd	高低半字节交换	$Rd(3 \sim 0) \leftarrow Rd(7 \sim 4), Rd(7 \sim 4) \leftarrow Rd(3 \sim 0)$	无	1
BSET	s	标志置位	$SREG(s) \leftarrow 1$	SREG(s)	1
BCLR	s	标志清零	$SREG(s) \leftarrow 0$	SREG(s)	1
BST	Rr, b	从寄存器将位赋给 T	$T \leftarrow Rr(b)$	T	1
BLD	Rd, b	将 T 赋给寄存器位	$Rd(b) \leftarrow T$	无	1
SEC		进位位置位	$C \leftarrow 1$	C	1
CLC		进位位清零	$C \leftarrow 0$	C	1
SEN		负标志位置位	$N \leftarrow 1$	N	1
CLN		负标志位清零	$N \leftarrow 0$	N	1
SEZ		零标志位置位	$Z \leftarrow 1$	Z	1
CLZ		零标志位清零	$Z \leftarrow 0$	Z	1
SEI		全局中断使能	$I \leftarrow 1$	I	1
CLI		全局中断禁用	$I \leftarrow 0$	I	1
SES		符号测试标志位置位	$S \leftarrow 1$	S	1



续表

指令	操作数	说 明	操 作	标 志	时钟数
位和位测试指令					
CLS		符号测试标志位清零	$S \leftarrow 0$	S	1
SEV		2 的补码溢出标志置位	$V \leftarrow 1$	V	1
CLV		2 的补码溢出标志清零	$V \leftarrow 0$	V	1
SET		SREG 的 T 置位	$T \leftarrow 1$	T	1
CLT		SREG 的 T 清零	$T \leftarrow 0$	T	1
SEH		SREG 的半进位标志置位	$H \leftarrow 1$	H	1
控制指令					
NOP		空操作		无	1
SLEEP		休眠		无	1
WDR		复位看门狗		无	1
BREAK		暂停只针对片内调试		无	

### 2.1.2 ATmega16 单片机的寻址方式

寻址是 ATmega16 单片机寻找操作数所在地址单元的过程，它有以下几种寻址方式。

- 直接单寄存器寻址。通过指令中的一些特定字段的内容来定位操作数，操作数的地址由指令的后 5 位来确定，可以寻址 32 个通用寄存器。
- 直接双寄存器寻址。与直接单寄存器寻址类似，但是它有两组 5 位寄存器地址单元，因此可以寻址 32 个通用寄存器中的两个。
- 直接 I/O 寻址。使用指令中的 D5 ~ D0 位来指定 I/O 的位置，同时还可以寻址一个寄存器单元作为目的地址或者源地址。
- 直接数据寻址。用于对 RAM 的操作，在该寻址方式中，一个字为操作数在 RAM 中的地址，一个字为指令，问 i 中包括目的或源寄存器地址。
- 带偏移量的间接数据寻址。以 Y 或 Z 寄存器中的变址和指令中的偏移量地址之和来决定操作数在 RAM 中的地址，同时指令中还含有目的或源寄存器的地址。
- 间接数据寻址。指令中给出的寄存器内容即为操作数的地址。
- 预先减 1 的间接数据寻址。与间接数据寻址类似，但是操作数的地址为指定寄存器的内容减 1，同时再把这个减 1 之后的值赋给指定的寄存器。
- 程序存储器常量寻址。这是访问程序存储器中常量的直接寻址方式，常量的地址由 Z 寄存器给出。
- 程序存储器间接寻址。以 Z 寄存器的内容为地址继续执行程序的寻址地址就是程序存储器的间接寻址方式，在这种寻址方式的指令执行之后，Z 寄存器的内容会代替程序指针的值。
- 程序存储器相关寻址。以当前 PC 值和指令中所包含的相关地址 K 值之和为地址继续执行程序的寻址方式，当指令执行之后，PC 值和 K 值相加，并且把两者的和送到 PC 中。



## 2.2 ATmega16 单片机 C 语言的数据类型、运算符和表达式

数据是 ATmega16 单片机操作的对象，是具有一定格式的数字或者数值。数据按照一定的数据类型进行的排列、组合和架构称为数据结构，ATmega16 的 C 语言支持的数据类型如表 2.2 所示。

表 2.2 ATmega16 的 C 语言支持的数据类型

数据类型	名称	长度	值域
基本类型	字符型 unsigned char, char	1B	0~255, -128~127
	整型 unsigned int, int	2B	0~65525, -32768~32767
	长整型 unsigned long, long	4B	0~4294967295, -2147483648~2147482647
	浮点型 float	4B	$\pm 1.176E.38E \sim \pm 3.40 + 38$ (6 位数字)
	双精度浮点型 double	8B	$\pm 1.176E.38E \sim \pm 3.40 + 38$ (10 位数字)
构造类型	数组		
	结构体		
	共用体		
	枚举		
指针类型		2~3B	存储空间, 最大 64KB
空类型			



### 注意

ATmega16 的 C 语言本身是不支持位变量和位操作的，但是某些编译器可以提供宏文件来支持位变量和位操作。

### 2.2.1 常量和变量

ATmega16 单片机的 C 语言的数据可以分为常量和变量。常量在程序执行过程中值不能发生变化，变量在程序执行过程中值可以改变；常量通常用#define 关键字来定义，而变量通常用变量名来表示，用户自己定义，它是一个起始字符为字符或者下划线，随后字符必须是字母、数字或者下划线的字符组合，变量在使用之前必须先定义。

### 2.2.2 算术运算、赋值、逻辑运算以及关系运算

在 ATmega16 单片机的 C 语言中，把用算术运算符和括号将运算对象连接起来的表达式称为算术表达式，运算对象包括常量、变量、函数、数组和结构等。在算术表达式中需要遵守一定的运算优先级，其规定为先乘除（余），后加减，最优先括号，同级别从左到右，与数学计算相同。

ATmega16 单片机的 C 语言一共支持 5 种算术运算符，如表 2.3 所示。

ATmega16 单片机的 C 语言值运算符包括普通赋值运算符和符合赋值运算符两种，普通的赋值运算符使用“=”，而符合赋值运算符是在普通赋值运算符之前加上其他二目运算符所构成的赋值符，使用赋值运算符连接的变量和表达式则可以构成赋值表达式。



表 2.3 算术运算符

运 算 符	意 义	说 明
+	加法运算或者正值符号	
-	减法运算或者负值符号	
*	乘法运算符号	
/	除法运算符号, 求整	5/2, 结果为 2
%	除法运算符号, 求余	5%2, 结果为 1

赋值运算涉及变量类型的转换, 可以分为 2 种: 一种是自动转换; 另一种是强制转换。自动转换是不使用强制类型转化符, 直接将赋值运算符右边表达式或者变量的值类型转化为左边的类型, 一般来说是从“低字节宽度”向“高字节宽度”转换, 表 2.4 为赋值中的自动类型转化。强制转换则是使用强制类型转化符来将一种类型转化为另外一种类型, 强制类型转化符号和变量类型相同。

表 2.4 赋值中的自动类型转化

类 型	说 明
浮点型和整型	浮点类型转化为整型时小数点部分被省略, 只保留整数部分; 反之只是把整型修改为浮点型
单、双精度浮点型	单精度转化为双精度时在尾部添 0, 反之进行四舍五入的截断操作
字符型和整型	字符型转化为整型时, 仅仅修改类型, 反之只保留整型的低八位

ATmega16 的 C 语言提供了 3 种逻辑运算符, 分别如下。

- 逻辑与: &&。
- 逻辑或: ||。
- 逻辑非: !。

使用逻辑运算符将表达式或者变量连接起来的表达式称为逻辑表达式。逻辑运算内部运算次序是, 先逻辑非后逻辑与和逻辑或, 相同等级为从左到右, 逻辑表达式的值为“真”或“假”, 在 ATmega16 单片机的 C 语言中使用“0”代表“假”, 使用“非 0”代表逻辑“真”, 但是逻辑运算表达式结果只能使用“1”来表示“真”。

ATmega16 单片机的 C 语言提供了 6 种关系运算符, 分别如下。

- 小于: <。
- 大于: >。
- 小于或等于: <=。
- 大于或等于: >=。
- 如果等于: ==。
- 如果不等于: !=。

使用关系运算符连接的表达式或者变量称为关系表达式, 关系运算符中前两种优先级高于后两种, 同等优先级下遵守从左到右的顺序, 关系运算式的运算结果是逻辑真“1”或者是逻辑假“0”。

### 2.2.3 自增减、复合和逗号运算

ATmega16 单片机 C 语言的自增减运算分别是使变量的值增加或者减少 1, 相当于“变





量 = 变量 + 1” 或者 “变量 = 变量 - 1” 操作，其应用形式是 “变量 ++”、“++ 变量”、“变量 --” 和 “-- 变量”。

复合运算是将普通运算符和赋值符号结合起来的运算，有两个操作数的运算符都可以写成 “变量、运算符 = 变量” 的形式，相当于 “变量 = 变量、运算符、变量”。

逗号运算的关键字为 “，”，其一般应用形式如 “表达式 1, 表达式 2, ..., 表达式 n”，逗号表达式按照从左到右的方式运算，整个表达式的值取决于最后一个表达式。

### 2.2.4 位运算

ATmega16 单片机 C 语言支持位操作，恰当的位操作能极大地方便用户编程，ATmega16 的位操作包括位逻辑运算和移位运算两种类型。

位逻辑运算包括位与、位或、位异或、位取反。

- 位与：关键字 “&”，如果两位都为 “1”，则结果为 “1”，否则为 “0”。
- 位或：关键字 “|”，如果两位其中有一个为 “1”，则结果为 “1”，否则为 “0”。
- 位异或：关键字 “^”，如果两位相等则为 “1”，否则为 “0”。
- 位取反：关键字 “~”，如果该位为 “1”，则取反后为 “0”；如果该位为 “0”，则该位取反后为 “1”。

移位运算包括左移位和右移位运算。

- 左移位：关键字 “<<”，将一个变量的各个位全部左移，空出来的位补 0，被移出变量的位则舍弃不要。
- 右移位：关键字 “>>”，操作方式相同，移动方向向右。

### 2.2.5 运算的优先级

表 2.5 为 ATmega16 单片机的 C 语言中运算符的优先级。

表 2.5 运算符优先级

优先级	关 键 字	说 明	运算次序
1	( ) [ ] → .	括号 下标运算，用于数组 指向结构成员，用于结构体 结构成员体，用于结构体	从左到右
2	! ~ ++ -- - (强制类型转换) * & sizeof	逻辑非运算符 按位取反运算符 自增运算符 自减运算符 负号运算符 类型转换运算符 指针运算符 取地址运算符 长度运算符	从右到左

续表

优先级	关 键 字	说 明	运算次序
3	* / %	乘法运算符 除法运算符 取余运算符	从左到右
4	+ -	加法运算符 减法运算符	从左到右
5	>> <<	右移运算符 左移运算符	从左到右
6	<, <=, >, >=	关系运算符	从左到右
7	==, !=	测试等于和不等运算符	从左到右
8	&	按位与运算符	从左到右
9	^	按位异或运算符	从左到右
10		按位或运算符	从左到右
11	&&	逻辑与运算符	从左到右
12		逻辑或运算符	从左到右
13	?:	条件运算符	从右到左
14	复合运算符		从右到左
15	,	逗号运算符	从左到右

## 2.3 ATmega16 单片机 C 语言的结构

为了根据不同的情况做出不同的控制动作，ATmega16 单片机的 C 语言和标准 C 语言一样，提供了控制流语句，通过不同的控制流语句的嵌套和组合可以控制 ATmega16 实现复杂的功能。控制流语句包括 if、else if、switch、while 等。

ATmega16 单片机的 C 语言的程序结构可以分为顺序结构、选择结构和循环结构，这三种结构可互相组合和嵌套，组成复杂的程序结构，完成相应的功能。

### 1. 顺序结构

顺序结构是最简单和基本的程序结构，程序从程序空间的低地址位向高地址位执行。

### 2. 选择结构

在选择结构中，程序首先测试一个条件语句，如果条件为“真”时执行某些语句，如果条件为“假”时执行另外一些语句。选择语句可以分为单分支结构以及多分支结构，多分支结构又包括串行多分支结构和并行多分支结构。选择语句构成了单片机判断和转移的基础，是模块化程序的重要组成部分，ATmega16 单片机 C 语言常用的选择语句有 if 语句、switch 语句，其中 if 语句有 if…else、if 和 else if 三种形式。

### 3. 循环结构

循环语句用于处理需要重复执行的代码块，在某个条件为“真”时，重复执行某些相同的代码块。循环语句一般由循环体（循环代码）和判定条件组成，ATmega16 单片机 C 语言常用的循环语句有 while 语句、do while 语句和 for 语句。



#### 4. break、continue 和 goto 语句

在循环语句执行过程中，如果需要在满足循环判定条件的情况下跳出代码块，可以使用 break、continue 语句，如果要从任意地方跳到代码的某个地方，可以使用 goto 语句。

## 2.4 ATmega16 单片机 C 语言的函数

ATmega16 单片机 C 语言支持把整个程序划分为若干个功能比较单一的小模块，通过模块之间的嵌套和调用来完成整个功能，这些具有单一功能的小模块称为函数，也可以称为子程序或者过程。ATmega16 单片机 C 语言的程序就是由一个个的函数构成的，从一个主函数开始执行，调用其他函数后返回主函数，进行其他的操作，最后从主函数中退出整个 ATmega16 单片机的 C 语言程序。

### 2.4.1 函数的定义、参数和返回值

函数按照定义形式可以分为无参数函数和有参数函数，其定义方法如下。

```
类型标识符  函数名( )           //无参数函数
{
    声明语句和代码块;
}

类型标识符  函数名(形式参数列表)   //有参数函数
{
    声明语句和代码块;
}
```

函数的值是在函数执行完成之后通过 return 语句返回给调用函数语句的一个值，返回值的类型和函数的类型相同，函数的返回值只能通过 return 语句返回。在一个函数中可以使用一个以上的 return 语句，但是最终只能执行其中的一个 return 语句。如果函数没有返回值，则使用 void 标志。

### 2.4.2 函数的调用

一般而言，函数调用有使用函数名调用、函数结果参与运算以及函数结果作为另外一个函数的实际参数三种调用方式，需要注意的是，函数在被调用之前必须首先被声明。

ATmega16 单片机 C 语言的函数支持嵌套和递归调用。

### 2.4.3 局部变量和全局变量

局部变量是在某个函数中存在的变量，也可以成为内部变量，它只在该函数内部有效。局部变量可以分为动态局部变量和静态局部变量，使用关键字 auto 定义动态局部变量，使用关键字 static 定义静态局部变量。

全局变量是在整个源文件中都存在的变量，又称为外部变量。全局变量的有效区间是从定义点开始到源文件结束，其间的所有函数都可以直接访问该变量，如果定义点之前的函数

需要访问该变量，则需要使用 `extern` 关键字对该变量进行声明，如果全局变量声明文件之外的源文件需要访问该变量，也需要使用 `extern` 关键字进行声明。

## 2.5 ATmega16 单片机 C 语言的数组和指针

数组是一组由若干个具有相同类型的变量所组成的有序集合。它一般被存放在内存中一块连续的存储空间，数组中的每一个元素都相继占有相同大小的存储单元。数组的每一个元素都有一个唯一的下标，通过数组名和下标可以访问数组的元素。构成数组的变量类型可以是基本的数据类型，也可以是用户自定义的结构、联合等类型。由整型变量组成的数组称为整型数组，由字符型变量组成的数组称为字符型数组，同理还有浮点型数组和结构型数组等。数组可以是一维的、二维的和多维的，其定义方式如下。

```
类型 数组名[size]           //一维数组
类型 数组名[sizeA][sizeB]   //二维数组
char c_Name[10]              //字符数组
```

当一个数组被创建时，ATmega16 单片机的 C 语言编译器就会在存储空间里开辟一个连续的区域用于存放该数组的内容。对于一维数组来讲，会根据数组的类型在内存中连续开辟一块大小等于数组长度乘以数组类型长度（即类型占有的字节数）的区域。

当 ATmega16 单片机的 C 语言定义一个变量后，编译器就给这个变量在内存中分配相应的存储空间。比如对于字符型（`char`）变量就会在内存中分配一个字节的内存单元，而对于一个整型（`int`）变量则会分配两个字节的内存单元。

假设程序中定义了 3 个整型变量 `i`、`j`、`k`，它们的值分别是 1、3、5，假设编译器将地址为 1000 和 1001 的两字节内存单元分配给了变量 `i`，将地址为 1002 和 1003 的两字节内存单元分配给了变量 `j`，将地址为 1004 和 1005 的两字节内存单元分配给了变量 `k`，则变量 `i`、`j`、`k` 在内存中的对应关系如图 2.1 所示。

在内存中变量名 `i`、`j`、`k` 是不存在的，对变量的存取都是通过地址进行的。而存取的方式又分为两种：一种是直接存取方式，如 `int x = i * 3`，这时读取变量 `i` 的值是直接找到变量 `i` 在内存中的位置，即地址 1000，然后从 1000 开始的两个字节中读取变量 `i` 的值再乘以 3 作为结果赋给变量 `x`；另一种方式是间接存取方式，在这种方式下变量 `i`

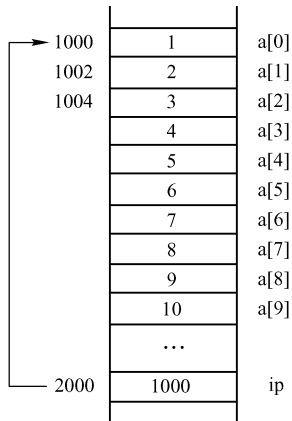


图 2.1 变量和内存地址的对应关系

的地址 1000 已经存在了某个地址（如 2000）中，这时当要存取变量 `i` 的值时，可以先从地址 2000 处读出变量 `i` 的地址 1000，然后再到 1000 开始的两个字节中读取变量 `i` 的值，这就是指针。

关于指针有两个重要的概念：变量的指针和指向变量的指针变量。

- 变量的指针。变量的指针就是变量的地址。如上面的例子中变量 `i` 的指针就是地址 1000。

- 指向变量的指针变量。在上例中，如果把用来存放变量  $i$  的地址的内存单元 2000 和一个变量关联，就像变量  $i$  关联地址单元 1000 一样，那么这个变量就称为指向变量  $i$  的指针变量。显然指针变量的值是指针（变量的地址）。

指针变量定义的一般形式为：

类型 \* 变量名

指针变量的引用是通过取地址运算符“&”来实现的，在定义完指针变量和引用之后就可以通过指针变量来对内存进行间接访问了。这时要用到指针运算符“\*”。其运算形式为：

\* 指针变量

其含义是指针变量所指向的变量的值，如果要将变量  $a$  的值赋给整型变量  $x$ ，就可以有两种访问方式了。

- 直接访问方式： $x = a$ 。
- 使用指针变量  $p$  进行间接访问： $x = *p$ ，此时程序先从指针变量  $p$  中读出变量  $a$  的指针（地址），然后从此地址的内存中读出变量  $a$  的值再赋给  $x$ 。

在 ATmega16 单片机的 C 语言中，指针和数组的关系十分密切，任何能由数组下标完成的操作也都可以用指针来实现，而且程序中使用指针可以使代码更紧凑、更灵活。



#### 注意

关于指针和数组的详细关联关系，读者可以自行参考其他 C 语言的资料，在此不多做叙述。

## 2.6 ATmega16 单片机 C 语言的自构造类型

构造新的数据结构是 C 语言的重要特点之一，结构体、联合体和枚举类型是 ATmega16 单片机的 C 语言支持用户自行构造的新数据类型。

### 2.6.1 结构体

结构体是一种或者多种类型变量的结合，这些变量可以是字符型、整型等，还可以是另外一个结构体，统称为结构体的成员。其构造方法为：定义为结构体数据类型的变量称为结构体变量，需要注意的是，只能对其中单个成员进行赋值和引用。

```
struct 结构名           //构造方法 1
{
    类型说明符 成员 1;
    类型说明符 成员 2;
    ...
    类型说明符 成员 n
}
结构名 变量名 1, 变量名 2, ...;
```





```
struct 结构名          //构造方法 2
{
    类型说明符 成员 1;
    类型说明符 成员 2;
    ...
    类型说明符 成员 n
} 变量名 1, 变量名 2, ...;

struct                //构造方法 3
{
    类型说明符 成员 1;
    类型说明符 成员 2;
    ...
    类型说明符 成员 n
} 变量名 1, 变量名 2, ...;
```

### 2.6.2 联合体

联合体又称为共用体，与结构体一样，它是一种构造类型，该类型用于在一块内存空间中存放不同类型的数据，该内存空间并不是所有类型数据所占用的内存大小的总合，而是由最大的变量空间决定的。其构造方法为：定义为联合体数据类型的变量称为联合体变量，它同样只能对其中单个成员进行赋值和引用。

```
union 结构名  //构造方法 1
{
    类型说明符 成员 1;
    类型说明符 成员 2;
    ...
    类型说明符 成员 n
}
结构名 变量名 1, 变量名 2, ...;

union 结构名  //构造方法 2
{
    类型说明符 成员 1;
    类型说明符 成员 2;
    ...
    类型说明符 成员 n
} 变量名 1, 变量名 2, ...;

union                //构造方法 3
{
    类型说明符 成员 1;
    类型说明符 成员 2;
    ...
```





```
类型说明符 成员 n  
}变量名 1,变量名 2,...;
```

需要注意的是，联合体变量在进行内存分配时是按照联合体变量成员中需要内存资源最大的变量分配的，在联合体变量占用的内存空间中始终只能保存联合体的一个成员有效数据，但是这个数据可以通过其他成员引用。

### 2.6.3 枚举

枚举数据类型同样也是构造类型，是某些整数型常量的集合，枚举类型数据变量的取值只能是这些常量中的一个，枚举类型变量的取值必须是定义中的整数值，其构造方式与结构体变量类似，需要说明的是，枚举类型一般用于替代变量的整数赋值。

```
enum 枚举名 //构造方法 1  
{  
    枚举值列表;  
};  
枚举名 变量 1,变量 2,...;  
enum 枚举名 //构造方法 2  
{  
    枚举值列表;  
}  
枚举名 变量 1,变量 2,...;
```

## 第3章 ATmega16 单片机的 ICC AVR 软件开发环境

AVR 系列单片机在嵌入式系统中得到了广泛的应用，包括 ATMEL 等在内的商业公司也为其开发了各种集成开发环境，本章简要介绍应用最为广泛的 ICC AVR 的使用方法，包括安装方法、工作界面、菜单和扩展关键字和库函数等。

### 3.1 ATmega16 单片机的软件开发环境

ATmega16 单片机常见的软件开发环境包括 AVR Studio、GCC AVR (WIN AVR)、ICC AVR、Code Vision AVR 等，这些开发环境的特点说明如下。

- AVR Studio。这是 ATMEL 公司提供的 AVR 单片机的汇编级开发环境，完全免费，并且集成了调试功能、AVR Prog 串行、并行下载功能和 JTAG ICE 仿真等功能；缺点是本身只支持汇编语言的开发，如果要支持 C 语言需要和其他编译环境集成。
- GCC AVR (WIN AVR)。GCC 本来是一个 Linux 上的 C 语言开发环境，当移植到 Windows 上用于 AVR 开发之后就变成了 WIN AVR，但是 WIN AVR 不是 IDE (集成开发环境)，需要用户使用 makefile 自己来定制 IDE，另外 WIN AVR 不支持 float 数据类型。
- ICC AVR。ICC AVR 是 ImageCraft 公司提供的一款基于 C 语言和汇编语言开发 AVR 单片机的 IDE 环境，集成度高，支持编译，并且可以和 AVR Studio 等环境联合调试。
- Code Vision AVR。Code Vision AVR 也是一个基于 C 语言的 IDE 开发环境，其特点是内置的程序生成向导比较友好，易于入门。

ICC AVR 集成开发环境是目前最常用的 AVR 单片机软件开发工具，其功能合适、使用方便、对各个型号的 AVR 单片机支持好，其主要有以下几个特点。

- 它是一个综合了编辑器和工程管理器的集成工作环境 (IDE)，集成度高、使用简单；
- 源文件全部被组织到工程之中，文件的编辑和工程的构筑也在这个环境中完成，编译错误会显示在状态窗口中，并且当点击错误时，光标自动跳转到有错误的那一行，便于用户进行修改和编译。
- 工程管理器可以生成 AVR 单片机直接使用的 .hex 格式文件，该格式的文件可被大多数编程器所支持，用于下载到芯片中。

### 3.2 安装 ICC AVR

ICC AVR 集成开发环境的安装步骤说明如下。

(1) 在光盘或者硬件中找到 ICC AVR 的安装程序 `iccavr6.31A.exe`（此处采用的是 `iccavr6.31A` 版本），双击运行，可以看到如图 3.1 所示的提示，一路点击“next”按钮之后，可以选择安装目录，如图 3.2 所示。

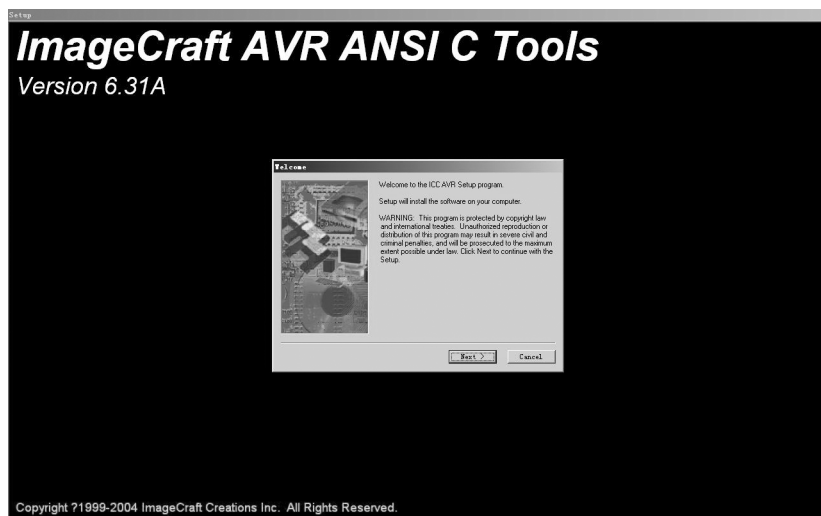


图 3.1 开始安装 ICC AVR

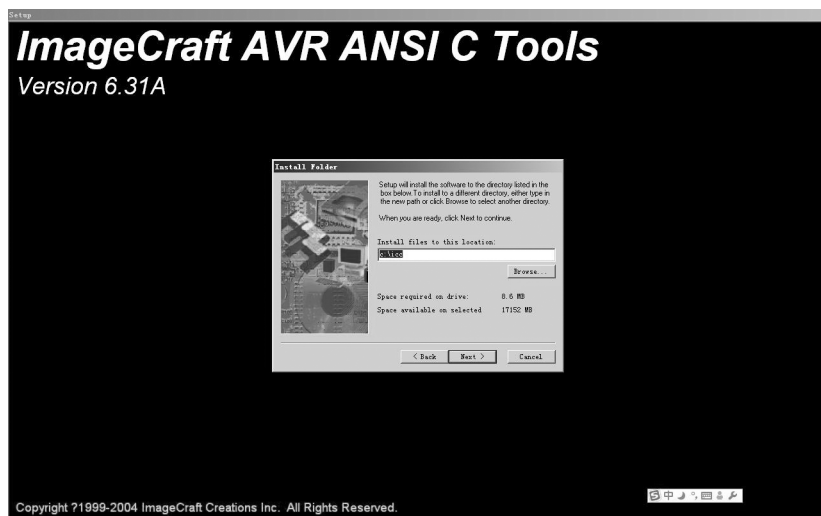


图 3.2 安装 ICC AVR，选择安装路径

(2) 点击“Next”按钮，然后选择安装的文件夹之后点击“install”按钮，出现如图 3.3 所示的安装进度，完成后点击“Finish”按钮即可。

(3) 安装完成之后，可以在“开始”菜单中找到对应的文件夹“ImageCraft Development Tools”，从中点击“ICC AVR”选项即可启动 ICC AVR 软件，如图 3.4 所示。

(4) 此时的 ICC AVR 软件还是一个仅可以使用 30 天的未注册版本，需要在启动软件之后在“Help/Register Software”中进行注册，方可获得完整的版本，如图 3.5 所示。

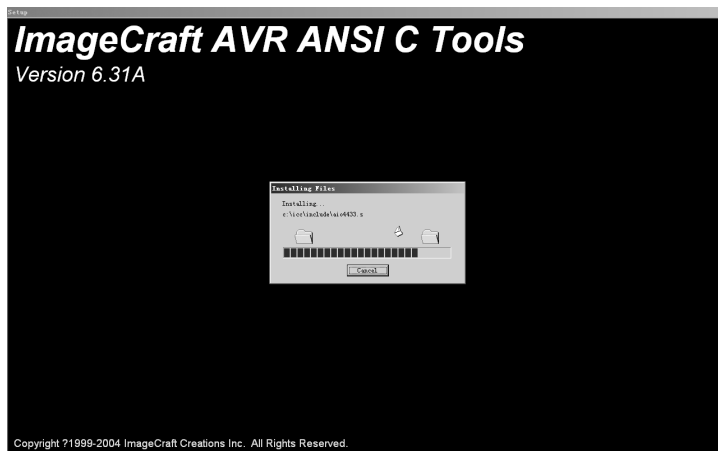


图 3.3 安装 ICC AVR, 进度

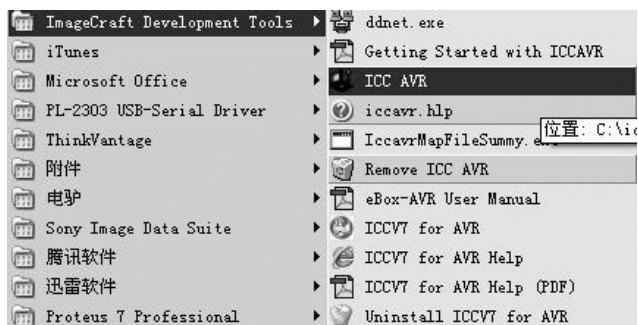


图 3.4 ICC AVR 软件

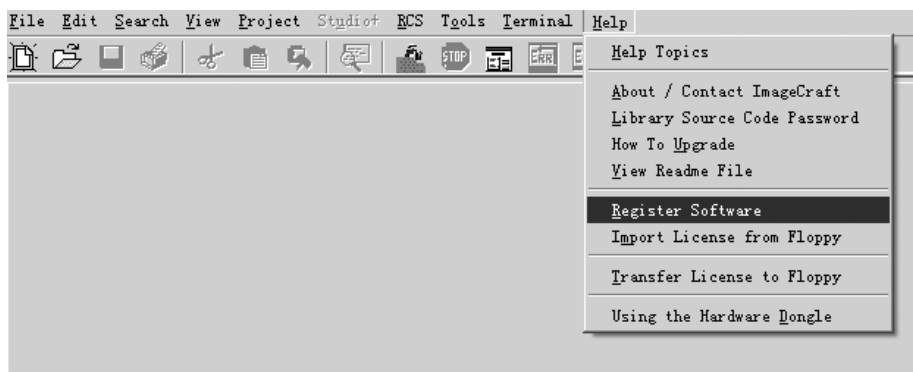


图 3.5 注册 ICC AVR 软件

### 3.3 ICC AVR 的工作界面

ICC AVR 集成开发环境提供了丰富的内部工具, 常用命令都具有快捷工具栏, 并且提供了 ICC AVR Application Builder 快速开发工具用于帮助用户快速建立一个应用工程文件。除了代码编辑窗口, ICC AVR 还提供了菜单栏、快捷工具栏、项目管理窗口、代码窗口、



目标文件窗口、输出窗口等观察窗口，其工作界面如图 3.6 所示。

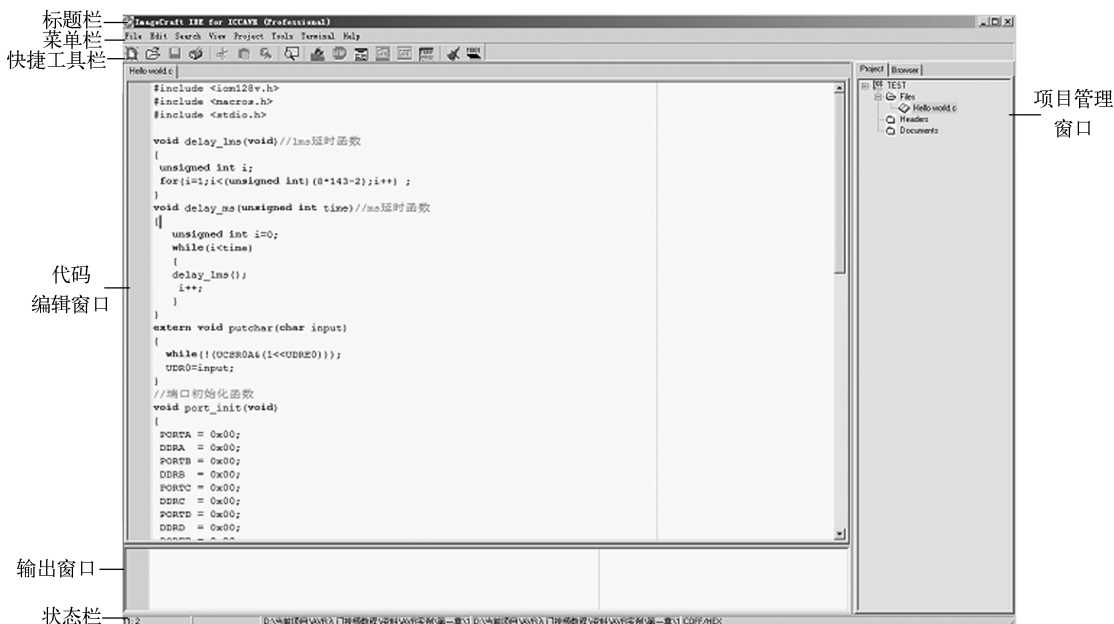


图 3.6 ICC AVR 的工作窗口

## 3.4 ICC AVR 的菜单栏和快捷工具栏

ICC AVR 提供了菜单栏和快捷工具栏以供用户进行操作，后者是菜单栏中的一些命令的快捷方式集合（本节内容基于 ICC AVR 7.22 版本）。

### 3.4.1 ICC AVR 的菜单栏

ICC AVR 的菜单栏包括 File、Edit、Search、View、Project、Tools、Terminal、Help 菜单项，下面详细介绍这些菜单的用途。

#### 1. File（文件）菜单

图 3.7 是 File（文件）菜单的示意图，该菜单包含了用于文件操作的相关菜单项，详细说明如下。

- New：新建一个文件。
- Reopen：打开历史文件，最近用过的历史文件将显示在右边的子菜单中。
- Open：打开一个已存在的文件。
- Reload：放弃全部的修改从磁盘或者备份中重新装载当前文件。
- Save：保存当前文件。
- Save as：将当前文件保存为另外的名称。
- Close：关闭当前文件。
- Compile File：把当前文件编译成目标文件、输出 .HEX 或者当前项目的启动文件。

- Save All Files: 保存所有打开的文件。
- Closs All Files: 关闭当前打开的所有文件。
- Print: 打印当前文件。
- Exit: 退出 ICC AVR IDE 环境。

## 2. Edit (编辑) 菜单

图 3.8 是 Edit (编辑) 菜单的示意图, 该菜单包含了编辑操作相关的菜单项, 其详细说明如下。

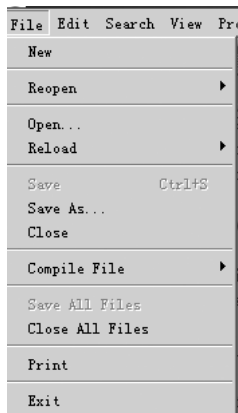


图 3.7 File 菜单

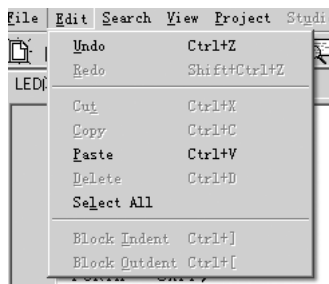


图 3.8 Edit 菜单

- Undo: 撤销上一次的修改。
- Redo: 恢复上一次撤销的操作。
- Cut: 把选择的内容剪切到粘贴板。
- Copy: 把选择的内容复制到粘贴板。
- Paste: 将粘贴板的内容粘贴在当前光标的位置。
- Delete: 删除所选择的内容。
- Select All: 选择当前文件的全部内容。
- Block Indent: 对当前选中的整块内容右移。
- Block Outdent: 对当前选中的整块内容左移。

## 3. Search (查找) 菜单

图 3.9 是 Search (查找) 菜单的示意图, 该菜单包含了用于查找操作的相关菜单命令, 其详细说明如下。

- Find: 在当前编辑窗口中寻找一个或者多个字符串, 可以使用 Match Case (区分大小写)、Whole Word (全字匹配)、Up/Down (往上或往下) 选项。
- Find in Files: 在当前所有打开的文件中或在当前工程的所有文件中或当前目录中的文件中寻找一个或者多个字符串, 可以使用 Case Sensitive (区分大小写)、Whole Word (全字匹配)、Regular Expression (寻找规则的表达式) 选项。
- Replace: 在当前编辑窗口中替换一个或者多个字符串。
- Search Again: 再次查找。

- Goto Line Number: 转到当前编辑窗口的指定行。
- Goto First Error: 定位到第一个错误。
- Goto Next Error: 定位到下一个错误。
- Add Bookmark: 添加书签。
- Delete Bookmark: 删除书签。
- Next Bookmark: 跳转到下一个书签。
- Goto Bookmark: 跳转到指定的书签。

#### 4. View (视图) 菜单

图 3.10 是 View (视图) 菜单的示意图, 该菜单包含了用于控制 ICC AVR 相关显示参数设置的操作, 其详细说明如下。

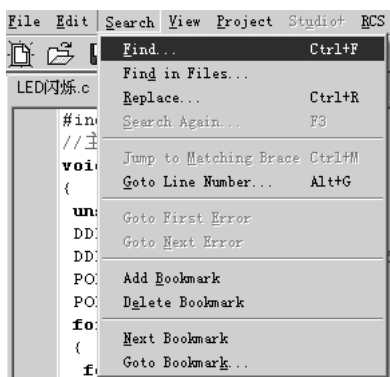


图 3.9 Search 菜单

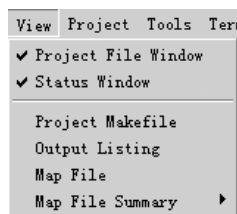


图 3.10 View 菜单

- Project File Window: 打开或者关闭项目组织窗口。
- Status Window: 打开或者关闭状态窗口。
- Project Makefile: 以只读方式打开当前项目的 makefile 文件。



图 3.11 Project 菜单

- Output Listing: 以只读方式打开当前项目的输出列表文件。
- Map File: 以只读方式打开当前项目的映射文件。
- Map File Summary: 打开当前项目映射文件的汇总, 可以选择 With Full Size Info (包括完整大小信息) 和 Without Size Info (不包括大小信息) 两个选项。

#### 5. Project (项目) 菜单

图 3.11 是 Project (项目) 菜单的示意图, 该菜单包含了用于控制 ICC AVR 项目设置相关参数的操作, 其详细说明如下。

- New: 创建一个新的工程项目。
- Open: 打开一个已经存在的工程项目。



- Open All Files: 打开选定工程的全部文件。
- Close All Files: 关闭当前全部打开的文件。
- Reopen: 重新打开一个最近打开过的工程项目。
- Make Project: 编译当前工程项目。
- Rebuild All: 重新编译当前工程项目。
- Add File(s): 添加一个文件到当前工程项目中。
- Remove Selected File(s): 从工程项目中删除选中的文件。
- Options: 打开工程项目编译选项菜单项。
- Clean Output Directory: 清除输出目录内容。
- Manual Sort Browser Window: 手工布置工程管理窗口的显示方式。
- Close: 关闭当前工程项目。
- Save As: 使用新名称保存当前工程项目。

Options 菜单是 Project 菜单中最重要的菜单项，用于设置当前项目的相关编译参数，包含 4 个菜单页，其中最常用的两个菜单页如图 3.12 和图 3.13 所示。

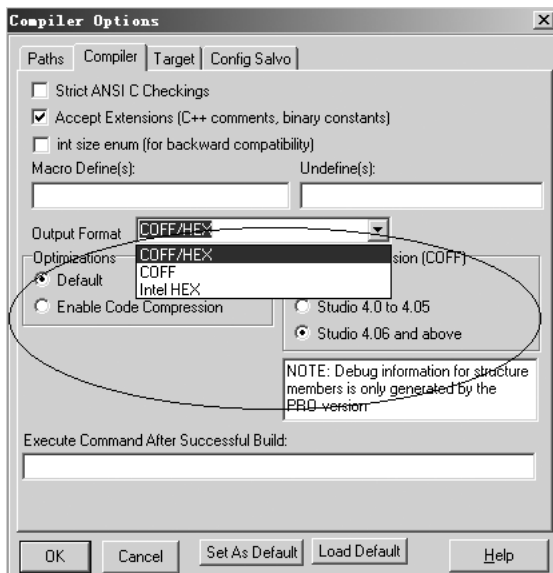
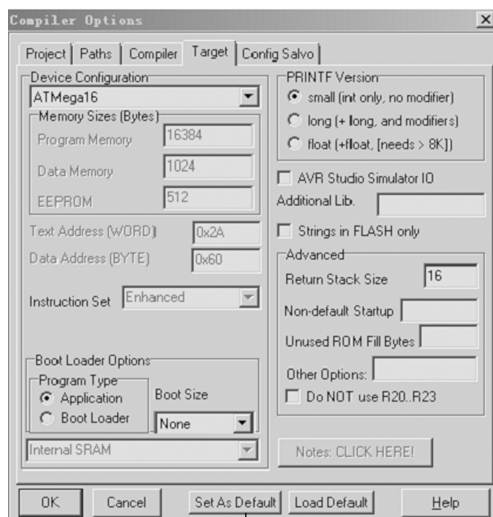


图 3.12 编译菜单项



设为默认值

图 3.13 目标菜单项

如图 3.12 所示，Options 菜单选项中的 Compiler 菜单页，常用的设置是选择项目的“Output Fomat”，通常情况下选择“COFF/HEX”即可，该选择生成项目对应的 .coff 文件和 .hex 文件，前者用于调试，后者用于对 ATmega16 编程。

如图 3.13 所示，Options 菜单选项中的 Target 菜单页，常用的设置是选择使用的具体器件、选择当前项目的类型是应用代码还是 Boot Loader 代码，以及选择 PRINTF 的数据类型，相关参数说明如下，还可以将其设置为默认值。

- small 或 basic: 只支持 %c、%d、%x、%X、%u 和 %s 格式。
- long: 支持 %ld、%lu、%lx、%IX。

- floating: 支持%f，该选项会占用很大的内存空间。

## 6. Tools（工具）菜单

图 3.14 是 Tools（工具）菜单的示意图，该菜单中包含了用于对 ICC AVR 的外部相关工具控制的操作，其详细说明如下。

- Environment Options: 环境和终端仿真器选项。
- Editor and Print Options: 编辑和打印选项选项。
- Mail Setup: 设置邮件信息，会弹出邮件信息设置对话框。
- In System Programmer: 在系统编程器，提供程序代码下载的应用，需要相应的下载硬件配合。
- AVR Calc: AVR 计算器，可以计算串口波特率、定时计数器的定时常数等（该工具在 64 位的 Windows 下可能无法启动）。
- Application Builder: 应用向导程序，可以生成硬件的初始化代码。
- Configure Tool: 添加外部工具到工具菜单。
- Run: 以命令行方式运行一个外部程序。

## 7. Terminal（终端）菜单

图 3.15 是 Terminal（终端）菜单的示意图，该菜单中包括了对 ICC AVR 中内部集成的终端工具进行控制的操作，其详细说明如下。

- Show Terminal Window: 打开终端窗口。
- Clear Window: 清除终端窗口的内容。
- Capture: 捕捉终端窗口的内容。

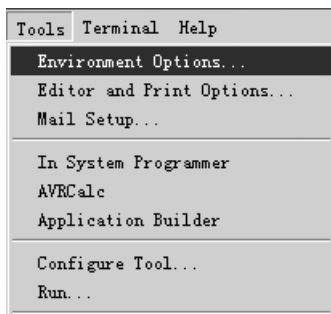


图 3.14 Tools 菜单

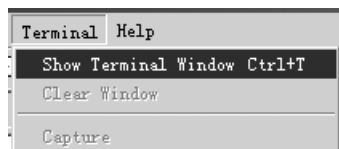


图 3.15 Terminal 菜单

## 8. Help（帮助）菜单

Help 菜单中是 ICC AVR 软件相关信息，帮助手册、升级以及注册帮助等操作的菜单，在此不再赘述。

### 3.4.2 ICC AVR 的快捷工具栏

ICC AVR 提供了如图 3.16 所示的快捷工具栏以供用户进行一些快速操作，其中每个按钮都对应了菜单栏中的菜单项，其详细说明如下。

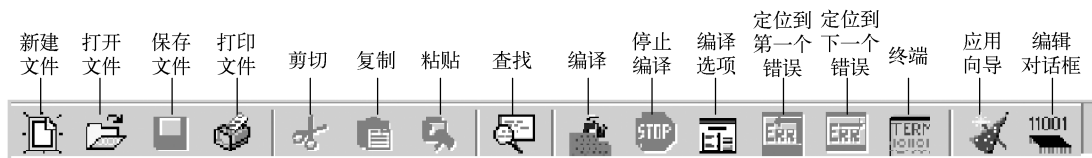


图 3.16 ICC AVR 的快捷工具栏

- 新建文件：新建一个文件，对应 File 菜单项中的 New 菜单命令。
- 打开文件：打开一个文件，对应 File 菜单项中的 Open 菜单命令。
- 保存文件：保存当前文件，对应 File 菜单项中的 Save 菜单命令。
- 打印文件：打印当前文件，对应 File 菜单项中的 Print 菜单命令。
- 剪切：将选择的内容剪切到粘贴板，对应 Edit 菜单项中的 Cut 菜单命令。
- 复制：将选择的内容复制到粘贴板，对应 Edit 菜单项中的 Copy 菜单命令。
- 粘贴：将粘贴板的内容粘贴到当前光标位置，对应 Edit 菜单项中的 Paste 菜单命令。
- 查找：打开查找对话框，对应 Search 菜单项中的 Find 菜单命令。
- 编译：对当前项目进行编译，对应 Project 菜单项中的 Make Project 菜单命令。
- 停止编译：停止对当前项目的编译过程，该按钮没有菜单命令，在编译进行过程中该按钮可用。
- 编译选项：打开当前项目编译选项对话框，对应 Project 菜单项中的 Options 菜单。
- 定位到第一个错误：定位到当前项目的第一个错误处，对应 Search 菜单项中的 Goto First Error 菜单命令，当编译完成出现错误后该按钮可用。
- 定位到下一个错误：定位到当前项目的下一个错误处，对应 Search 菜单项中的 Goto Next Error 菜单命令，当编译完成出现错误后该按钮可用。
- 终端：打开终端窗口，对应 Terminal 菜单项中的 Show Terminal Window 菜单命令。
- 应用向导：打开应用向导对话框，对应 Tools 菜单项中的 Application Builder 菜单命令。
- 编程对话框：打开在线编程器对话框，对应 Tools 菜单项中的 In System Programmer 菜单命令。

### 3.5 ICC AVR 的扩展关键字

ICC AVR 除了支持标准的 C 语言关键字之外，还支持一些扩展关键字，用于编译和诸如中断之类的特殊操作，这些关键字说明以及用法如下。

#### 3.5.1 中断关键字

```
#pragma interrupt_handler <name> : <vector number>
```

“interrupt\_handler” 关键字必须在函数之前定义，用于说明该函数是中断操作函数，编译器会在中断操作函数中生成中断返回指令 `reti` 来代替普通返回指令 `ret`，并且保存和恢复函数所使用的全部寄存器，并且会根据中断向量号 `vector number` 生成中断向量地址。



### 3.5.2 非挥发寄存器关键字

```
#pragma ctask <func1> <func2>...
```

“ctask”关键字指定了函数使用非挥发寄存器来保存和恢复代码，其典型应用是在 RTOS 实时操作系统中让 RTOS 核直接管理寄存器。

### 3.5.3 数据段关键字

```
#pragma data: <data>
```

“data”关键字用于改变数据段名称，使其与命令行选项相适应，该关键字在分配全局变量至 EEPROM 中时必须被使用。

## 3.6 ICC AVR 的文件

除了用户的设计文件之外，ICC AVR 还提供了库函数文件和启动文件以供用户配合使用。

### 3.6.1 ICC AVR 的常用文件类型

ICC AVR 的常用文件类型扩展名如下。

- .c: C 语言源文件。
- .s: 汇编语言源文件或者 C 语言源文件在编译器编译时产生的汇编输出文件。
- .h: C 语言头文件。
- .prj: 工程项目文件，该文件保存项目的相关信息。
- .a: 库文件，可以由几个库封装在一起，其中 libcavr.a 是一个由 ICC AVR 提供的，包含了标准 C 和 AVR 特殊程序调用的基本库，如果某个库被引用，ICC AVR 内置的连接器会将其连接到对应的模块或文件中。用户也可以创建或修改一个符合自己需要的库。
- .o: 汇编文件汇编产生的目标文件，多个目标文件可以连接成一个可执行文件。
- .hex: INTEL HEX 格式文件，是用户代码对应的机器代码。
- .eep: INTEL HEX 格式文件，是 EEPROM 的初始化数据。
- .cof: COFF 格式输出文件，用于在如 ATMEL 的 AVR Studio 等环境下进行程序调试。
- .lst: 列表文件，此文件中列举出了目标代码对应的最终地址。
- .mp: 内存映像文件，包含了用户程序中有关符号信息及其所占内存大小的信息。
- .cmd: NoICE 2.xx 的调试命令文件。
- .noi: NoICE 3.xx 的调试命令文件。
- .dbg: ICC AVR 的调试命令文件。



### 3.6.2 ICC AVR 的库函数文件

ICC AVR 也提供了现成库函数以供用户调用,使用这些函数可以大大地提高代码编写效率,在使用这些库函数之前必须先引用如下所列的对应头文件。

- `io*.h`: I/O 寄存器操作函数的头文件。
- `macros.h`: 宏和定义声明的头文件。
- `assert.h`: 宏声明的头文件。
- `ctype.h`: 字符类型函数头文件。
- `float.h`: 浮点数原型头文件。
- `limits.h`: 数据类型的大小和范围头文件。
- `math.h`: 浮点运算函数头文件。
- `stdarg.h`: 变量参数表头文件。
- `stddef.h`: 标准定义头文件。
- `stdio.h`: 标准输入输出 I/O 函数头文件。
- `stdlib.h`: 内存分配函数的标准库头文件。
- `string.h`: 字符串处理函数头文件。

### 3.6.3 ICC AVR 的启动文件

ICC AVR 提供的启动文件可以用于对 ATmega16 进行初始化工作,包括清除 RAM 数据等操作,ICC AVR 的连接器会自动将启动文件连接到用户程序之前,将标准库 `libcavr.a` 和用户的程序相连接。启动文件根据目标芯片的不同在 `crtavr.o` 和 `crtatmega.o` 中间选择一个。

启动文件中定义了一个全局符号“`_start`”,它也是用户程序的起始执行点,启动文件的功能如下。

- 初始化硬件和软件堆栈指针。
- 从 `idata` 区复制初始化数据到直接寻址数据区 `data` 区。
- 将 `bss` 区全部初始化为零。
- 调用用户代码的主函数 `main()`。
- 定义一个退出点,如果用户主函数 `main()` 一旦退出,系统将进入这个退出点执行无限循环操作。
- 定义了复位向量。

在 ICC AVR 中可以有多个启动文件,可以在工程选项对话框中很方便地直接指定一个启动文件加入用户的工程,但是必须指定启动文件的绝对路径或启动文件必须位于工程选项库路径所指定的目录中。用户可以自行修改和使用新的启动文件,其详细操作步骤如下。

- (1) 进入 ICC AVR 安装路径。
- (2) 找到 `crtavr.s` 文件。
- (3) 用 IDE 打开 `crtavr.s` 文件。
- (4) 修改相关信息。

- (5) 选择编译到目标文件创建一个新的 crtavr.o。
- (6) 把 crtavr.o 复制到 ICCAR 的库目录。



#### 注意

如果使用的目标芯片是 ATmega 系列, 则应该用 “crtatmega” 代替 “crtavr”。

### 3.7 “Hello World!” ——ICC AVR 的应用实例

由于 ICC AVR 自带项目管理器, 所以用户不需要在项目管理上花费过多的精力, 只需要按照以下步骤操作即可建立一个属于自己的项目。

- (1) 启动 ICC AVR, 建立工程文件并且选择器件。
- (2) 建立源文件、头文件等相应的文件。
- (3) 将工程需要的源文件、头文件、库文件等添加到工程中。
- (4) 修改启动代码并且设置工程相关选项。
- (5) 编译并且生成 HEX/COFF 或者 LIB 文件。

例 3.1 是一个在 ICC AVR 中建立一个工程文件通过 ATmega16 的串口输出 “Hello World!” 字符串的实例。

#### 【例 3.1】建立 “Hello World” 工程文件

(1) 建立工程文件: 启动 ICC AVR, 选择菜单 Project/New, 输入 “Exam301HelloWorld” 并且保存文件。

(2) 点击 “File/New”, 建立一个新文件, 并且将该文件命名为 “HelloWorld.c” 后保存, 并且在文件中输入如下的代码。

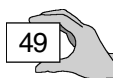
```
#include <iom16v.h>
#include <macros.h>
#include <stdio.h>
//1ms 延时函数
void delay_1ms(void)
{
    unsigned int i;
    for(i = 1; i < (unsigned int)(8 * 143 - 2); i++);
}
//ms 延时函数
void delay_ms(unsigned int time)
{
    unsigned int i = 0;
    while(i < time)
    {
        delay_1ms();
        i++;
    }
}
```



```

}
//串口发送函数
extern void putchar( char inputdata)
{
    while( !( UCSRA&( 1 << UDRE) ));
    UDR = inputdata;
}
//端口初始化函数
void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0x00;
    PORTB = 0x00;
    DDRB = 0x00;
    PORTC = 0x00;
    DDRC = 0x00;
    PORTD = 0x00;
    DDRD = 0x00;
}
//串口初始化函数,9600b/s
void uart0_init( void)
{
    UCSRB = 0x00;                                //在设置波特率的同时关闭使能
    UCSRA = 0x00;
    UCSRC = BIT( URSEL) | 0x06;
    UBRRL = 0x33;
    UBRRH = 0x00; hi
    UCSRB = 0x48;
}
//串口发送中断处理函数
#pragma interrupt_handler uart0_tx_isr:iv_USART0_TXC
void uart0_tx_isr( void)
{
    //字符已发送完成
    delay_ms( 50);                                //延时
    printf( " hello world!");                      //输出 hello world!
}
//ATmega16 初始化函数
void init_devices( void)
{
    CLI();
    port_init();
}

```



```

uart0_init();
MCUCR = 0x00;
GICR  = 0x00;
TIMSK = 0x00;                      //时钟中断源
SEI();                             //重新启用中断
//初始化了所有的外围端口
}
//主函数
void main(void)
{
    init_devices();
    delay_ms(50);                   //延时
    while(1)                        //进入主循环
    {
    }
}

```



### 注意

如果 ICC AVR 是 6.13 之后的版本，则需要将 ICC AVR 安装目录下 include 文件夹的 STDIO.H 头文件中 putchar 函数的声明修改为 void putchar(char)，这是因为 6.13 版本之后的 ICC AVR 将不再提供 putchar 函数，需要用户自行添加。

(3) 在项目管理器中点击右键，在弹出菜单中选择“Add File(s)”，将“HelloWorld.c”文件加入项目中，如图 3.17 所示。



图 3.17 将文件添加到项目



### 注意

需要将项目中所用的所有文件，包括 .h 文件、.c 文件、.lib 文件等都添加到对应的工程项目中去。

(4) 在项目管理窗口中点击右键，选择“Options”，在弹出的项目属性菜单中设置“Exam301HelloWorld”项目的相关属性，在本项目中将“Device Configuration”（芯片配置）修改为“ATmega16”，其他选项都维持不变，如图 3.18 所示。



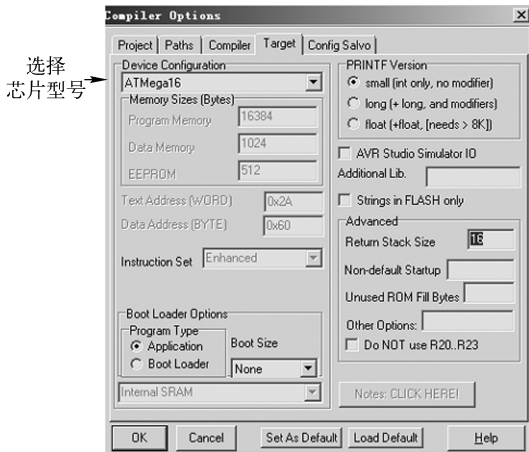


图 3.18 选择项目的目标器件

(5) 点击菜单“Project/Make Project”对当前项目进行编译，此时 ICC AVR 的输出窗口如图 3.19 所示。

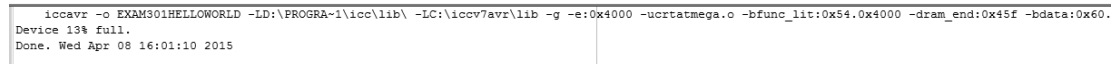


图 3.19 ICC AVR 的输出窗口

此时一个 ICC AVR 的项目已经编译完成，在文件编辑过程中，常常会由于编辑时的粗心或者其他原因出现一些语法上的错误，ICC AVR 会自动检测这种错误并且在编译时报告，通过双击输出框中的提示信息，ICC AVR 会自动将光标移动到提示信息所对应的错误信息所在行（定位不一定准确），修改代码中的错误代码之后再次编译项目，则会得到正确的编译结果。

# 第4章 ATmega16 单片机的硬件开发和 Proteus 硬件仿真环境

本章介绍 ATmega16 单片机的硬件系统开发流程和常用工具，以及 Proteus 硬件仿真环境使用方法，Proteus 是英国 Labcenter Electronics 公司出品的 EDA 工具软件，它可以对 ATmega16 单片机应用系统进行仿真，并且支持和 ICC AVR 进行联合调试，本章介绍其基本使用方法以及和 ICC AVR 的联合调试方法。

## 4.1 ATmega16 单片机的硬件系统开发流程和开发工具

ATmega16 单片机的硬件系统开发是一个综合性的过程，其涉及了硬件和软件的结合使用，还涉及了如 ISP 下载器等具体的硬件开发工具。

### 4.1.1 ATmega16 单片机的硬件系统开发流程

一个完整的 ATmega16 项目开发包括硬件和软件两个环节，其中硬件开发流程如图 4.1 所示，包括硬件逻辑设计、电路原理图设计、电路 PCB 图设计等步骤，详细描述如下。

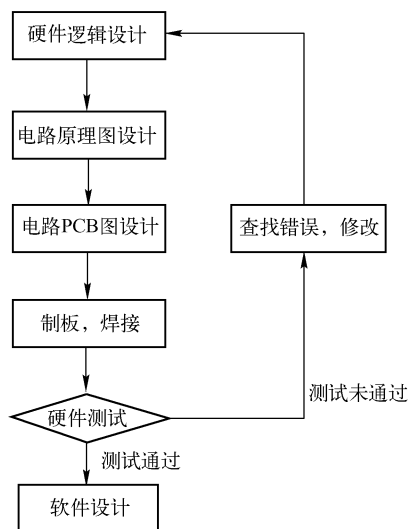


图 4.1 ATmega16 硬件系统设计流程

(1) 硬件逻辑设计。根据 ATmega16 单片机以及外部器件的相关使用方法进行硬件逻辑上的设计，包括电气连接、地址分配等。

(2) 电路原理图设计。利用电路图设计相关软件设计 ATmega16 单片机系统的电路原理图，为下一步 PCB 图设计做准备。

(3) 电路 PCB 图设计。在电路原理图设计的基础上进行 ATmega16 单片机系统的 PCB 图设计，这是 ATmega16 单片机系统硬件设计的最关键步骤，不仅仅要求电气物理逻辑连接正确，期间还需要有设计技巧和规则需要遵循。

(4) 制板、焊接。制板是指将 PCB 图制作成电路板实体的过程，一般在电路板厂中完成，焊接是指在电路板制作完成之后将对应的元器件用焊锡固定到电路板上并且使得其电气物理连接到一起的过程。

(5) 硬件测试。在电路板制作并且焊接完成之后对硬件部分进行测试，确定其有没有



逻辑或者电气上的错误，如果有则需要返回第一个步骤进行修改。

(6) 软件设计。在硬件测试通过之后即可以开始进行 ATmega16 单片机系统的上的软件开发，让软/硬件配合实现系统的设计目标。

### 4.1.2 ATmega16 单片机的硬件开发工具

ATmega16 单片机需要借助开发工具才能实现系统的开发设计，常见的 ATmega16 开发工具主要包括运行软件开发环境的计算机、ISP 编程器、万用表、示波器、万用开发板等，其中 ISP 编程器、数字万用表、数字示波器是比较常用的工具。

#### 1. ISP 编程器

ISP 编程器是将 ICC AVR 编译生成的十六进制 (.hex) 文件下载到 ATmega16 中的工具，最常见的 ISP 编程器是广州双龙电子公司生产的 SL - USBISP 编程器，其外形如图 4.2 所示。



图 4.2 SL - USBISP 编程器

USBISP 编程器的一头使用 USB 连接线和 PC 的 USB 口连接，在 PC 上安装好对应的下载软件之后将编程器连接，PC 上会出现“找到新硬件”的提示，并且自动安装好相关的驱动，PC 端上的软件界面如图 4.3 所示。

USBISP 编程器和 ATmega16 采用 10 芯/6 芯的 IDC 接口连接，其接口定义如图 4.4 所示。



图 4.3 USBISP 的 PC 端软件

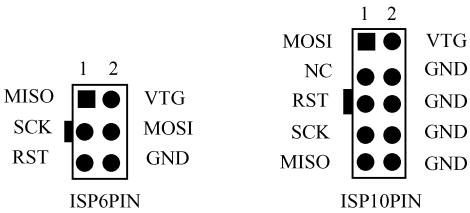


图 4.4 USBISP 和 ATmega16 的接口

当连接好 USBISP 编程器、PC 和 ATmega16 之后即可对 ATmega16 进行在线编程，其他具体的操作步骤可以参考该编程器的说明文档。

#### 2. 数字万用表

数字万用表是一种多功能、多量程的测量仪表，一般万用表可测量直流电流、直流电压、交流电压、电阻和音频电平，还可以测量交流电流、电容量、电感量及半导体的一些

参数，在 ATmega16 单片机系统开发中，常常使用万用表来检测电路的物理电气连接和引脚逻辑电平，图 4.5 是常用的数字万用表实物图。

### 3. 数字示波器

数字示波器是一种用途十分广泛的电子测量仪器。它能把肉眼看不见的电信号变换成看得见的图像，利用示波器能观察各种不同信号幅值随时间变化的波形曲线，还可以用它测试各种不同的电量，如电压、电流、频率、相位差、调幅值等。在 ATmega16 单片机系统开发中，常常使用万用表来观察相关的波形，图 4.6 是常用的数字示波器实物图。



图 4.5 数字万用表



图 4.6 数字示波器

## 4.2 Proteus 应用基础

Proteus 是一个基于 ProSPICE 混合模型仿真器的、完整的嵌入式系统软硬件设计仿真平台，它由 ISIS 和 ARES 两大应用功能软件组成，前者是一个原理图输入软件，用于电路原理设计和仿真，后者则用于 PCB 电路图布线。Proteus 可以实现从原理图设计、ATmega16 编程、ATmega16 应用系统仿真到应用系统 PCB 设计的流程化工作，其具体功能模块组成结构如图 4.7 所示。

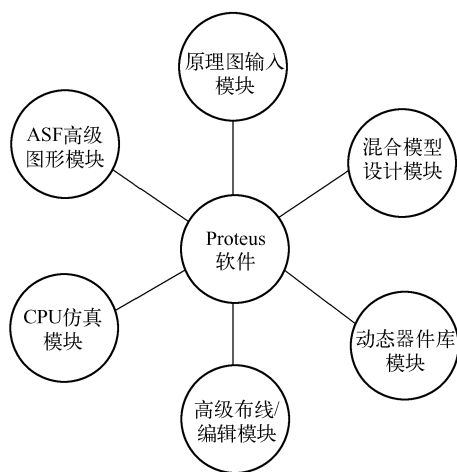


图 4.7 Proteus 的功能组成



### 注意

本书仅介绍 Proteus ISIS 原理图输入仿真软件部分，关于 ARES 读者可以自行参考其他资料。

### 4.2.1 Proteus 的界面和支持的文件

Proteus 的运行界面如图 4.8 所示，可以看到其由预览窗口、编辑窗口、器件显示窗口等，以及菜单栏、快捷工具栏、工具箱、仿真工具栏等组成。

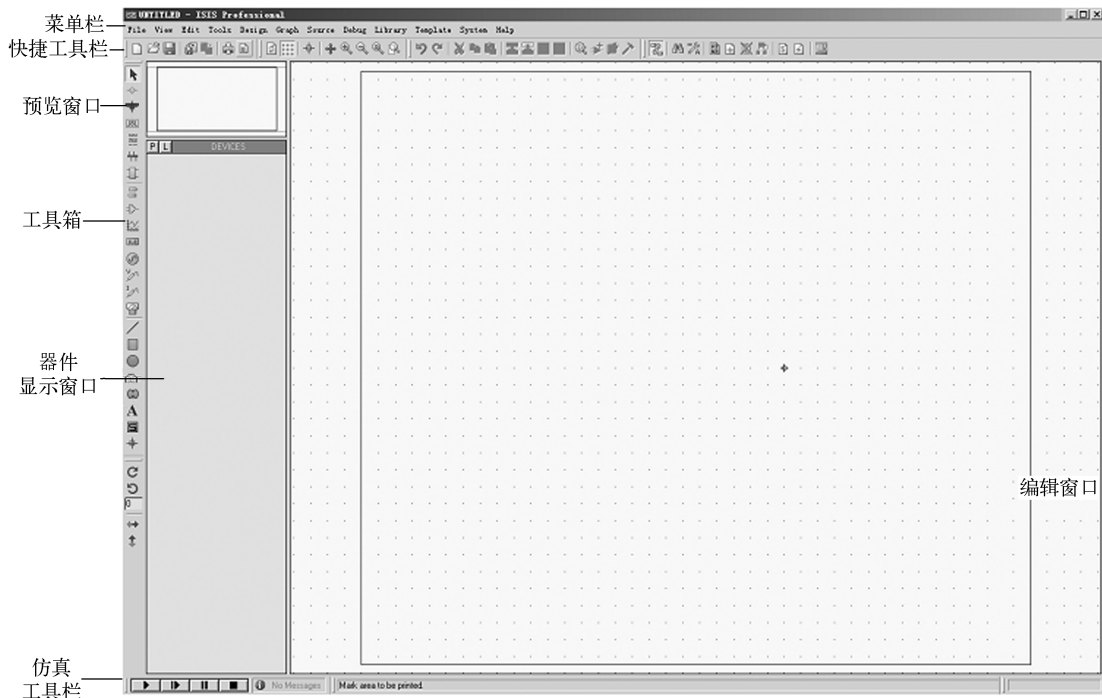


图 4.8 Proteus 的运行界面

Proteus 的窗口可以分为编辑窗口、预览窗口、器件显示窗口三大区域，每个窗口都有自己独特的作用，其详细说明如下。

- 编辑窗口（Editing Window）。编辑窗口用于放置元器件，进行连线，绘制原理图，输出运行和仿真结果等，这是 Proteus 的主要操作和显示区域。
- 预览窗口（Overview Window）。预览窗口用于显示当前的图纸布局和正在操作的器件相关情况。
- 器件显示窗口（Components Window）。器件显示窗口用于在当前项目加载的各个器件的相关情况，包括器件名称、引脚分布等。

除了三个常用的窗口，Proteus 还有菜单栏、快捷工具栏、工具箱、仿真工具栏等常用的辅助操作栏，其详细说明如下。

- 菜单栏。提供相应的操作菜单，单击任何一个菜单项后都会弹出子菜单项。
- 快捷工具栏。提供相应的操作快捷按钮，单击后会启动对应的快捷操作。
- 工具箱。提供诸如虚拟仪器等工具的启动操作，单击后回启动对应的工具。
- 仿真工具栏。提供启动仿真、暂停仿真等操作的快捷按钮。

Proteus 支持如下文件格式。

- .DSN: Design Files，这是 Proteus ISIS 的设计文件。
- .DBK: Backup Files，这是 Proteus ISIS 的备份文件。
- .SEC: Section Files，这是 Proteus ISIS 的部分电路存盘文件。
- .MOD: Module Files，这是 Proteus ISIS 的器件仿真模式文件。

- .LIB: Library Files, 这是 Proteus ISIS 的器件库文件。
- .SDF: Netlist Files, 这是 Proteus ISIS 的网络列表文件。

## 4.2.2 Proteus 的菜单

图 4.9 是 Proteus 的菜单项示意图, 它提供了文件、试图、编辑、工具、设计、图形、源设置、调试、库元件、模板、系统设置和帮助共 12 个菜单项。

File	View	Edit	Tools	Design	Graph	Source	Debug	Library	Template	System	Help
文件 菜单	视图 菜单	编辑 菜单	工具 菜单	设计 菜单	图形 菜单	源设置 菜单	调试 菜单	库元件 菜单	模板 菜单	系统设 置菜单	帮助 菜单

图 4.9 Proteus 的菜单项

### 1. File 菜单

Proteus 的 File 菜单项主要用于对文件的操作, 包括新建、加载、保存、打印等选项, 分为设计文件操作、选择区域操作、打印操作、最近打开的文件、退出五部分, 如图 4.10 所示, 其详细说明如下 (括号中为对应的快捷键)。



图 4.10 Proteus 的 File 菜单项

- New Design: 新建一个文件。
- Open Design (Ctrl + O): 打开已有的文件。
- Save Design (Ctrl + S): 保存当前文件。
- Save Design As: 将当前文件保存为另外一个文件。
- Save Design As Template: 将当前文件保存为模板。
- Windows Explorer: 打开 Windows 的文件管理器。

- Import Bitmap: 导入图片文件。
- Import Section: 导入部分选中的文件。
- Export Section: 导出选中的文件, 该选项在平时是灰色的, 只有当前部分图形被选中时变为有效。
- Export Graphics: 将导出的文件保存为图片, 提供了如图 4.11 所示的 5 种格式, 包括 BMP、EMF、DXF、EPS 和 HGL 文件。
- Mail To: 将当前文件作为邮件发送。
- Print: 打印当前文件。
- Printer Setup: 设置打印机。
- Printer Information: 打印信息。
- Set Area: 设置答应区间。
- Exit: 退出 Proteus ISIS。

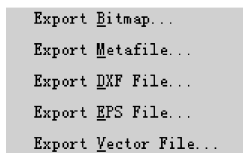


图 4.11 Proteus 导出支持的图片格式

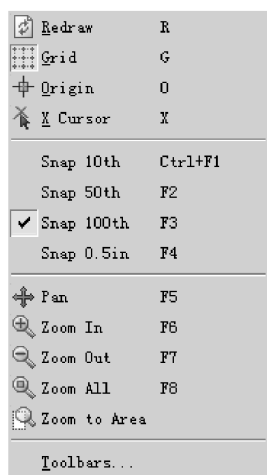


图 4.12 Proteus 的 View 菜单项

## 2. View 菜单

Proteus 的 View 菜单项主要用于设置 Proteus 相关显示内容, 包括图形刷新、坐标选择、放大缩小、是否显示快捷菜单栏等操作, 如图 4.12 所示, 其详细说明如下 (括号中为对应的快捷键)。

- Redraw (R): 刷新设计图纸, 会去掉图纸上无效的相关图形。
- Grid (G): 打开或者关闭图纸上的参考坐标点。
- Origin (O): 设置图纸的坐标原点。
- X Cursor (X): 修改图纸的 X 坐标。
- Snap 10th (Ctrl + F1): 选择坐标点密度为 10th。
- Snap 50th (F2): 选择坐标点密度为 50th。
- Snap 100th (F3): 选择坐标点密度为 100th。
- Snap 0.5in (F4): 选择坐标点密度为 500th。
- Pan (F5): 以当前鼠标位置为中心显示图纸。
- Zoom In (F6): 放大图纸。
- Zoom Out (F7): 缩小图纸。
- Zoom All (F8): 将图纸缩小到显示全部。
- Zoom to Area: 显示某个区域, 按住鼠标右键用于选择显示的区域。
- Toolbars: 用于打开或者关闭对应的快捷菜单栏, 如图 4.13 所示, 分别提供了 File Toolbar (文件相关快捷菜单栏)、View Toolbar (显示相关快捷菜单栏)、Edit Toolbar (编辑相关快捷菜单栏)、Design Toolbar (设计相关快捷菜单栏)。

## 3. Edit 菜单

Proteus 的 Edit 菜单项通常用于对 Proteus 设计图的全部或者部分区域进行操作, 包括取



消刚刚完成的操作或者重复刚刚取消的操作，剪切、复制等，如图 4.14 所示，其详细说明如下（括号中为对应快捷键）。

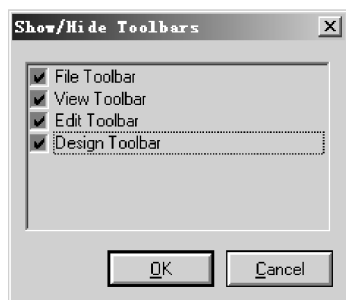


图 4.13 Toolbars 显示控制对话框

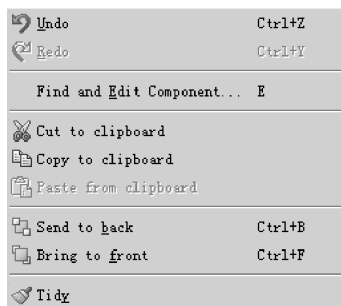


图 4.14 Proteus 的 Edit 菜单项

- Undo (Ctrl + Z): 取消刚刚完成的操作。
- Redo (Ctrl + Y): 重做刚刚取消的操作。
- Find and Edit Component (E): 查找和编辑器件。
- Edit Object Under Cursor (Ctrl + E): 编辑鼠标选中的目标。
- Cut to clipboard: 将选中部分剪切到粘贴板。
- Copy to clipboard: 将选中部分复制到粘贴板。
- Paste from clipboard: 将粘贴板的内容复制到当前文件。
- Send to back (Ctrl + B): 选中目标到后台，当有多层图形叠加时有效。
- Bring to front (Ctrl + F): 选中目标到前台，多层图形叠加时有效。
- Tidy: 清理工件列表中没有使用的器件。

#### 4. Tools 菜单

Tools 菜单项提供了对 Proteus 的电路图的某些自动操作，如自动添加器件标号，自动标注器件，自动生成图纸的材料清单，自动生成网络表等操作，如图 4.15 所示，其详细说明如下（括号中为对应的快捷键）。

- Real Time Annotation (Ctrl + N): 实时标注，当该选项被选中，在放置一个新的器件时，Proteus ISIS 会自动给该器件加上编号。
- Wire Auto Router (W): 自动连线，当该选项被选中时，当鼠标移动到一个引脚时会自动产生一个连线提示。
- Search and Tag (T): 搜索标签。
- Property Assignment Tool (A): 属性编辑工具。
- Global Annotator: 统一编号，用于对多张图纸组成的工程文件中的器件使用统一的编号，当选定时会弹出如图 4.16 所示的对话框，用于选择作用范围（Whole Design: 整个工程；Current Sheet: 单张图纸）、编号方式（Total: 全部相同；Incremental: 增量）、Initial Count（初始化数值设置）。



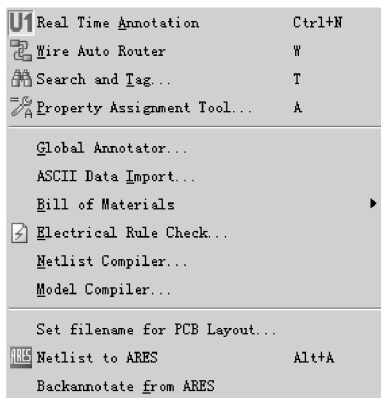


图 4.15 Proteus 的 Tools 菜单项

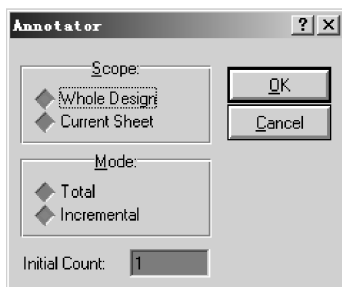


图 4.16 Global Annotator 设置对话框

- ASCII Data Import: ASCII 数据导入。
- Bill of Materials: 生成材料清单。
- Electrical Rule Check: 电气规则检查。
- Netlist Compiler: 生成网络表。
- Model Compiler: 模式编译。
- Set filename for PCB Layout: 设置对应的 PCB 层名称。
- Netlist to ARES (ALT + A): 从网络表生成电路板图。
- Backannotate from ARES: 从电路板返回标准信息。

## 5. Design 菜单

Proteus 的设计菜单项主要包括了 Proteus 对工程文件以及当前图纸的属性进行操作以及切换的相关命令, 如图 4.17 所示, 其详细说明如下 (括号中为对应的快捷键)。

- Edit Design Properties: 编辑工程属性, 用于编辑当前整个工程的属性, 点击后弹出如图 4.18 的对话框, 可以设置工程的名称、路径、作者、编号, 并且记录设计日期, 同时还可以设置工程的网络表相应属性。

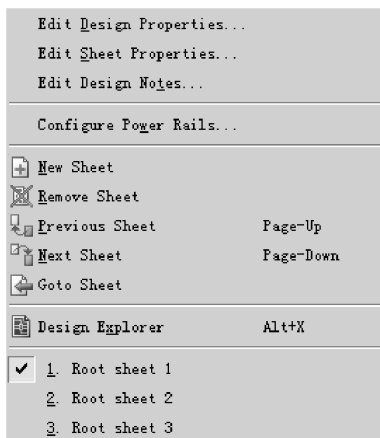


图 4.17 Proteus 的 Design 菜单项



图 4.18 设置工程项目属性对话框

- **Edit Sheet Properties**: 设置当前文件属性，同样会弹出相应的对话框。
- **Edit Design Notes**: 该命令会调出一个对话框用于记录设计者的一些注释。
- **Configure Power Rails**: 用于配置电源的相关隐含值，该命令会弹出如图 4.19 所示的对话框，在该对话框中可以找到所有设计中所有的电源列表，可以对其进行相应设计，例如，图 4.19 中则将该设计中所有的  $V_{CC}$  和 +5V 都连接到了一起。
- **New Sheet**: 添加一张新的图纸，和菜单命令 New design 不同，这是在同一个项目 design 下新建了一张图纸，常常用于一个较大的工程文件中的分模块设计。
- **Remove Sheet**: 删除当前图纸。
- **Previous Sheet (Page - Up)**: 上一张图纸。
- **Next Sheet (Page - Down)**: 下一张图纸。
- **Goto Sheet**: 切换到指定的图纸，此时会弹出如图 4.20 的对话框以供用户指定需要切换到的图纸。

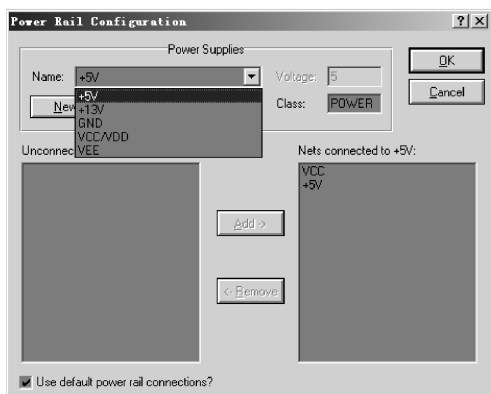


图 4.19 配置电源的相关隐含值

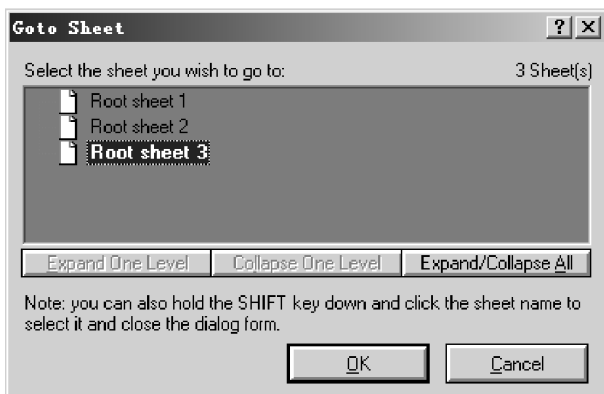


图 4.20 切换到指定图纸

- **Design Explore (Alt + X)**: 设计管理器，用于显示设计过程中的一些相关信息。
- 当前项目中的文件在 Design 菜单的最下方列出，点击对应的文件名即可在这些文件之间进行切换。

## 6. Graph 菜单

Proteus 的 Graph 菜单主要用于仿真操作，具有编辑仿真图形，添加仿真曲线及仿真图形、查看日志、导出数据、清除数据和一致性分析等功能，如图 4.21 所示，其详细说明如下（括号中为对应的快捷键）。

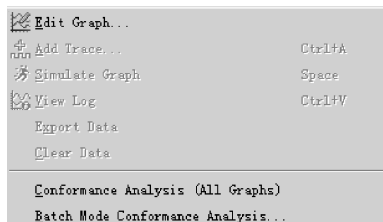


图 4.21 Proteus 的 Graph 菜单项

- **Edit Graph**: 编辑仿真图形。
- **Add Trace (Ctrl + A)**: 添加仿真曲线。
- **Simulate Graph (Space)**: 对图形进行仿真操作。
- **View log (Ctrl + V)**: 查看日志。
- **Export Data**: 输出仿真数据。

- **Clear Data**: 清除仿真数据。

- Conformance Analysis (All Graphs): 对所有图形进行仿真。
- Batch Mode Conformance Analysis: 批量仿真操作。

## 7. Source 菜单

Proteus 的 Source 菜单项, 主要用于对 Proteus 中需要驱动代码的器件设置相应的驱动源, 主要包括添加/删除源文件, 定义代码生成工具, 设置外部文本编辑器和编译等操作, 如图 4.22 所示, 其详细说明如下 (括号中为对应的快捷键)。

- Add/Remove Source files: 添加或者移除源文件, 点击后会弹出如图 4.23 所示的对话框, 其中 Target Processor 用于设置对应的处理器 (一个设计文件中有多个处理器存在的情况), Code Generation Tool (代码编辑工具, 但是仅支持 51 系列单片机的汇编语言), Source Code Filename (源代码文件)。

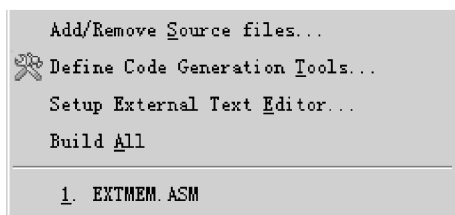


图 4.22 源文件菜单项

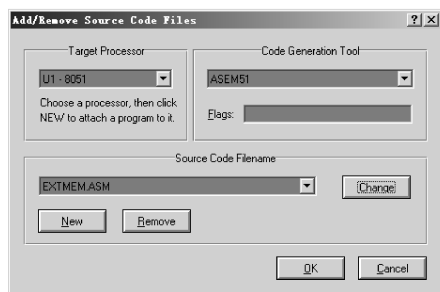


图 4.23 添加或者移除源文件

- Define Code Generation Tools: 定义代码产生工具, 用于设置源代码编译相关的一些选项。
- Setup External Text Editor: 设置外挂文本编辑工具。
- Bulid All: 编译当前所有的源文件。

图 4.21 中最下方带数字标号部分是项目当前涉及的源文件列表, 通过点击对应的源文件名可以在如图 4.24 所示编辑器中编辑对应的文件, 该编辑器支持对代码的修改、编辑和保存等操作。

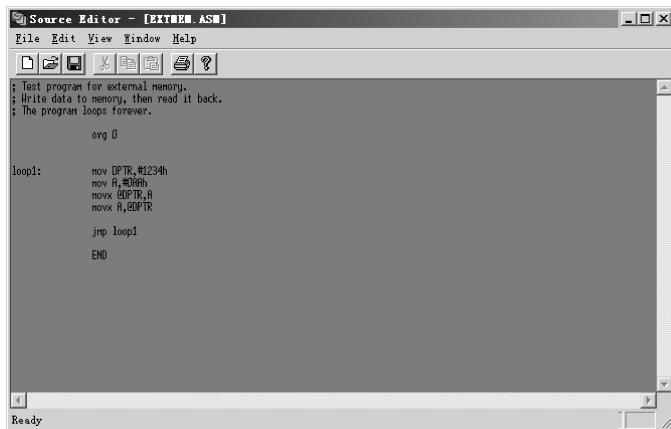


图 4.24 Proteus 的源文件编辑窗口

## 8. Debug 菜单

Proteus 的 Debug 菜单项主要用于在 Proteus 中进行调试操作，包括启动调试、执行仿真、单步运行、断点设置和重新排布弹出窗口等功能，如图 4.25 所示，其详细说明如下（括号中为对应的快捷键）。

- Start/Restart Debugging (Ctrl + F12)：启动或者重新启动调试。
- Pause Animation (Pause)：暂停调试。
- Stop Animation (Shift + Pause)：停止调试。
- Execute (F12)：调试执行。
- Execute Without Breakpoints (Alt + F12)：全速执行，不考虑断点。
- Execute for Specified Time：指定时间执行。
- Step Over (F10)：不进入子函数内部的调试。
- Step Into (F11)：跟踪调试，进入子函数内部。
- Step Out (Ctrl + F11)：从当前子程序中跳出。
- Step To (Ctrl + F10)：程序执行到指定位置。
- Reset Popup Windows：复位弹出窗口。
- Reset Persistent Mode Data：复位固定模式数据。
- Configure Diagnostics：配置相关的诊断信息，点击后会弹出如图 4.26 所示的对话框，用于对 Proteus 系统以及当前调试的处理器进行配置（如图 4.26 所示为 8052 处理器）。

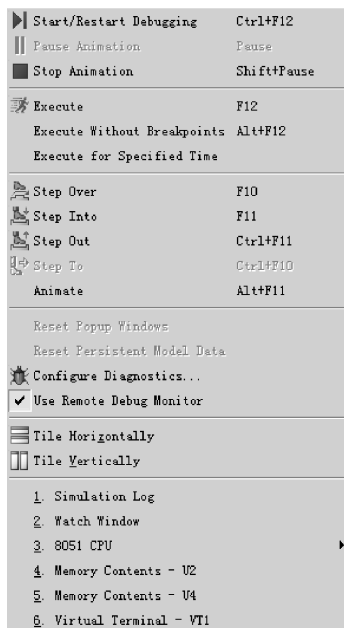


图 4.25 Proteus 的 Debug 菜单项

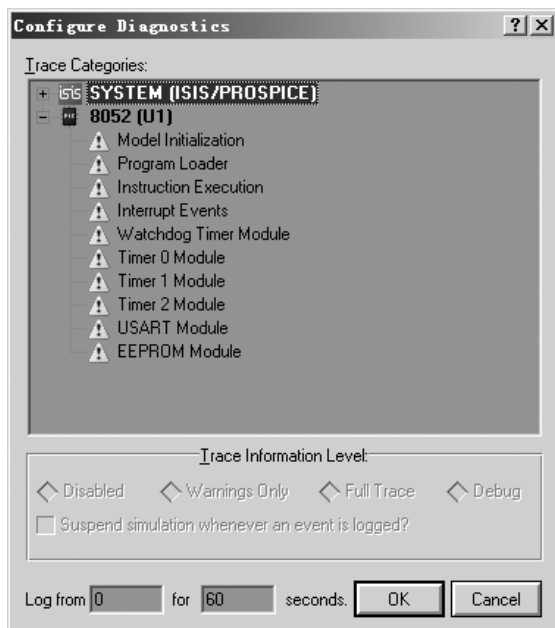


图 4.26 信息配置窗口

- Use Remote Debug Monitor：启动或者关闭用户远程调试窗口。

- Title Horizontalloy: 水平放置窗口。
- Title Vertically: 垂直放置窗口。

如图 4.25 所示最下方带数字部分是当前仿真的相关窗口，包括日志窗口、观察窗口、CPU 窗口等，通过点击这些窗口的名称可以在其间切换。

### 9. Library 菜单

Proteus 的 Library 菜单用于对 Proteus 自带的库元件以及用户自己引入的库元件进行管理，包括选择元器件及符号、制作元器件及符号、设置封装工具、分解元件、编译库、自动放置库、校验封装和调用库管理等操作，如图 4.27 所示，其详细说明如下（括号中为对应的快捷键）。

- Pick Device/Symbol (P): 从已有的器件库中复制器件符号。
- Make Device: 生成一个器件。
- Make Symbol: 生成一个符号。
- Packaging Tool: 器件封装工具。
- Decompose: 排列库中的元件。
- Compile to Library: 编译到库元件。
- Autoplace Library: 自动放置库。
- Verify Packaging: 验证库元件封装。
- Library Manager: 库管理器。



图 4.27 库菜单项

### 10. Template 菜单

Proteus 的 Template 菜单主要用于 Proteus 中的相关风格设置，包括设置图形格式、文本格式、设计颜色以及连接点和图形等，如图 4.28 所示，其详细说明如下。

- Goto Master Sheet: 转到当前项目的主图纸。
- Set Design Defaults: 设置图纸的默认值，主要用于设置相关选项的颜色、字体等，弹出如图 4.29 的对话框。



图 4.28 模板菜单项

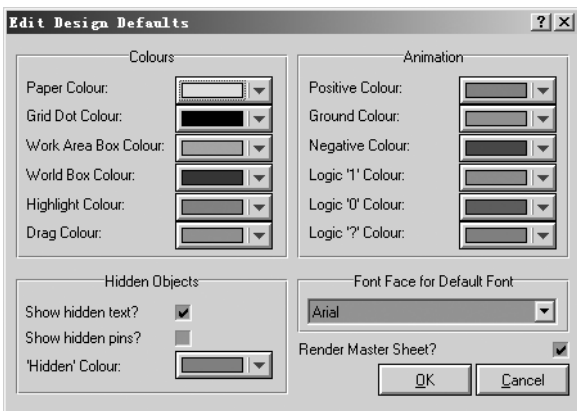


图 4.29 设置图纸的默认值

- **Set Graphics Colours:** 设置图纸的颜色、背景颜色，General Appearance（总体外观，包括图形的外轮廓线颜色、背景颜色、图纸标题等）、Analogue Traces（模拟信号的信号线颜色），Digital Traces（数字信号的信号线颜色，包括标准、总线、控制线、阴影等），如图 4.30 所示。

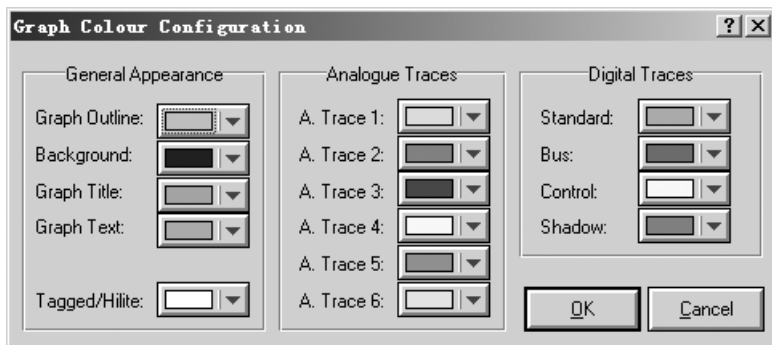


图 4.30 设置图纸和背景的颜色

- **Set Graphics Styles:** 设置图形风格，点击后会弹出如图 4.31 所示的对话框，主要用于设置元件（COMPONENT）、引脚（PIN）、端口（PORT）等的线风格（Line style）、宽度（Width）、颜色（Colour）、填充风格（Fill Attributes）等，在右侧的 Sample 区域内可以看到当前设置的效果。

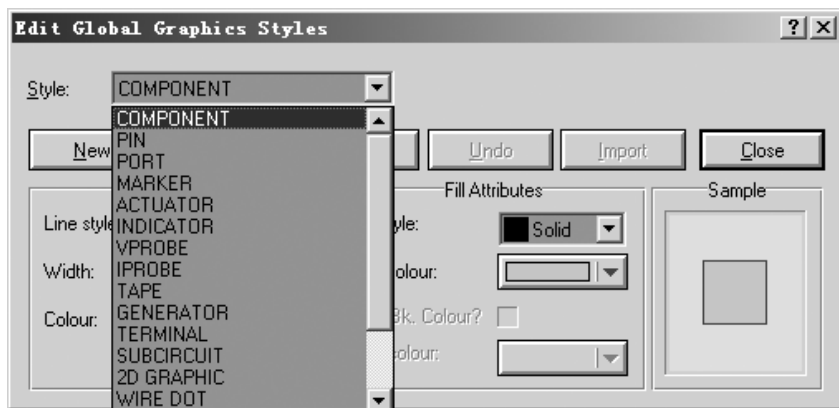


图 4.31 设置图形风格

- **Set Text Styles:** 设置文本风格，点击后会弹出如图 4.32 所示的对话框，主要用于设置器件名称（COMPONENT ID）、器件值（COMPONENT VALUE）、器件属性（PROPERTIES）的字体（Font face）、字体大小（Height）、颜色（Color）、效果（Effects）等，同样可以在下方的 Sample 区看到当前的设置效果。
- **Set Graphics Text:** 设置图形文本，点击后会弹出如图 4.33 所示的对话框，主要用于设置 2D 图形文字的字体（Font face）、文本的位置（Text Justification）、效果（Effects）、字体大小（Character Sizes）等。

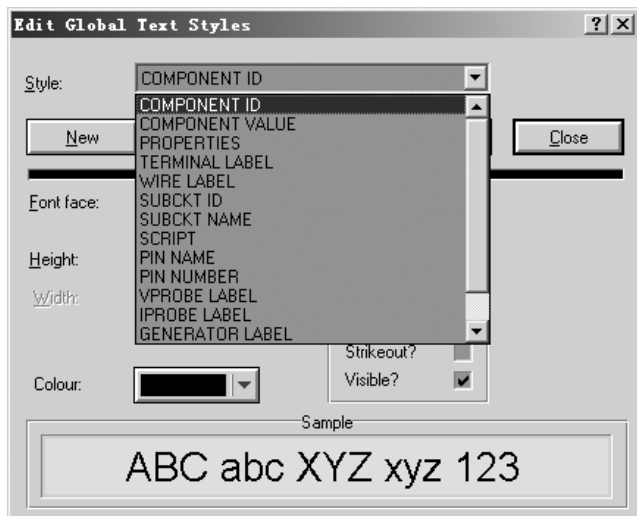


图 4.32 设置文本风格

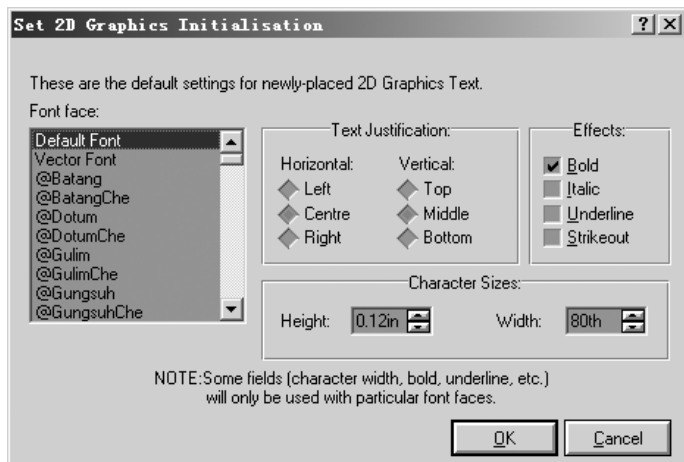


图 4.33 设置图形文本

- **Set Junction Dots:** 设置节点大小，点击后会弹出如图 4.34 所示的对话框，用于设置在原理图形中节点的大小（Size）和风格（Shape）。
- **Load Styles from Design:** 从项目中导入设计风格，该命令可以打开一个已经存在的设计文件并且将其风格应用于当前项目中。

## 11. System 菜单

Proteus 的 System 菜单用于对 Proteus 的相关参数进行设置，包括系统环境、路径、图纸尺寸、标注字体、热键以及仿真参数和模式等，如图 4.35 所示，其详细说明如下。

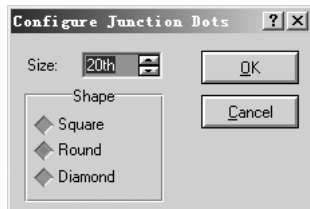


图 4.34 设置交叉点

- **System Info:** 显示当前系统信息, 包括 Proteus 的版本号、注册日期、操作系统的相关信息等。
- **Check for Updates:** 检查升级信息。
- **Text Viewer:** 打开文本浏览器。
- **Set BOM Scripts:** 设置器材清单的输出格式, 提供了 HTML、ASCII、紧凑型 CSV 和普通型 CSV 类型文件的输出格式。
- **Set Environment:** 设置相关环境参数, 点击后会弹出如图 4.36 所示的对话框, 主要用于自动保存时间 (Autosave Time)、Number of Undo Levels (可撤销的操作数据)、相关菜单栏提示延时 (Tooltip Delay), 在 File 菜单项下显示的最近使用的文件数目 (Number of filenames on File menu) 等。

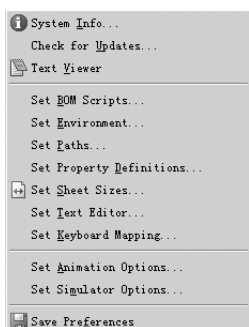


图 4.35 Proteus 的 System 菜单项

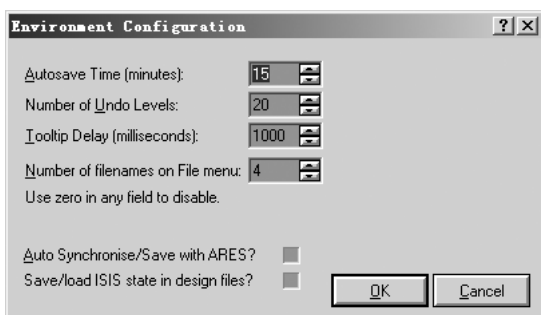


图 4.36 设置环境参数

- **Set Paths:** 设置相关的库文件和仿真文件的默认路径, 点击后会弹出如图 4.37 所示的对话框, 主要用于设置模板路径 (Template folders)、库文件路径 (Library folders)、仿真元件库文件 (Simulation Model and Module Folders) 等。

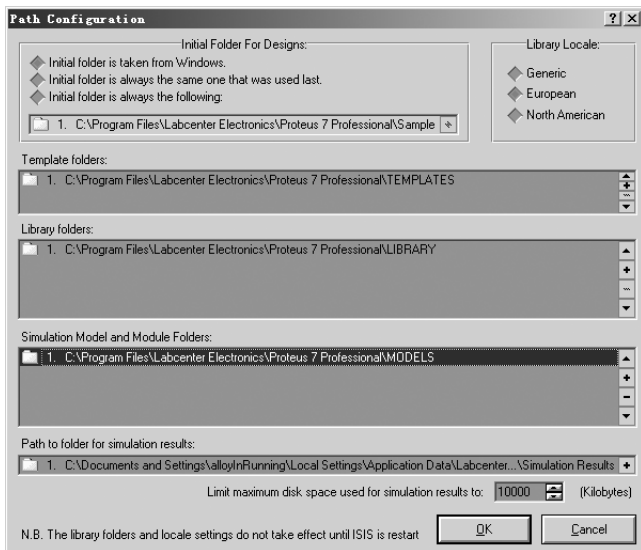


图 4.37 设置相关路径



- Set Property Definitions: 设置属性定义。
- Set Sheet Sizes: 设置图纸尺寸, 点击后会弹出如图 4.38 所示的对话框, 用于设置 Proteus 中默认的相关图纸尺寸大小, 包括 A0 ~ A4。
- Set Text Editor: 设置文本编辑器中的字体大小颜色风格等。
- Set Keyboard Mapping: 设置 Proteus 的快捷键。
- Set Animation Options: 设置仿真时的赋值参数, 点击后会弹出如图 4.39 所示的对话框, 主要用于设置仿真的速度 (Simulation Speed)、仿真的显示选项 (Animation Options)、电压和电流的范围 (Voltage/Current Ranges)。

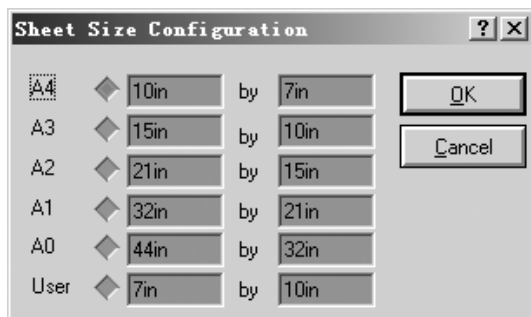


图 4.38 设置图纸尺寸

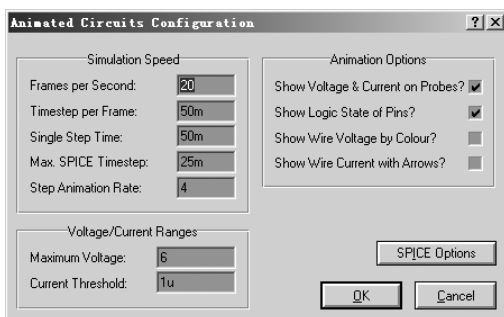


图 4.39 设置仿真的赋值参数

- Set Simulator Options: 设置仿真参数。
- Save Preferences: 保存当前设置的参数。

## 12. Help 菜单

Proteus 的 Help 菜单项用于给用户提供关于 Proteus 的相关操作信息, 包括版权信息、Proteus ISIS 学习教程和示例等, 如图 4.40 所示, 其详细说明如下。

- ISIS Help: 打开 Proteus ISIS 的基本帮助菜单。
- Proteus VSM Help: 打开到 Proteus ISIS 的 VSM 仿真操作说明。
- Proteus VSM SDK: 打开如何在 Proteus ISIS 下进行 VSM 仿真的操作说明。
- Sample Designs: 打开 Proteus ISIS 自带的示例所在文件夹。
- Stop Press: Proteus ISIS 的版本升级说明。
- About ISIS: Proteus ISIS 的相关版权说明。

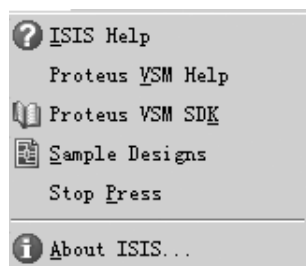


图 4.40 Proteus 的 Help 菜单项

### 4.2.3 Proteus 的快捷工具栏和工具箱





Proteus ISIS 的快捷工具可以分为菜单项下方的快捷工具栏和左侧的工具箱两部分, 如图 4.8 所示, 它为用户提供了一个快速操作的通道。

## 1. 快捷工具栏

Proteus 的快捷工具栏位于菜单项的下方，主要为用户提供文件（File Toolbar）、视图（View Toolbar）、编辑（Edit Toolbar）和设计（Design Toolbar）相关的快捷方式，可以通过 View 菜单项中的 Toolbar 选项进行关闭或者打开全部或者部分，其详细说明如下。

- 新建一个设计按钮：对应 File 菜单项的 New Design。
- 打开设计按钮：对应 File 菜单项的 Open Design。
- 保存当前设计按钮：对应 File 菜单项的 Save Design。
- 导入部分文件按钮：对应 File 菜单项的 Import Section。
- 输出部分文件按钮：对应 File 菜单项的 Export Section。
- 打印按钮：对应 File 菜单项的 Print。
- 标记输出区域按钮：用于标记需要输出的区域。
- 刷新显示按钮：对应 View 菜单项的 Redraw。
- 显示坐标按钮：对应 View 菜单项的 Grid。
- 设置坐标原点按钮：对应 View 菜单项的 Origin。
- 切换到坐标原点按钮：对应 View 菜单项的 Pan。
- 放大按钮：对应 View 菜单项的 Zoom In。
- 缩小按钮：对应 View 菜单项的 Zoom Out。
- 显示整张图纸按钮：对应 View 菜单项的 Zoom All。
- 区域显示按钮：对应 View 菜单项的 Zoom to Area。
- 撤销操作按钮：对应 Edit 菜单项的 Undo。
- 重做操作按钮：对应 Edit 菜单项的 Redo。
- 剪切按钮：对应 Edit 菜单项的 Cut to ClipBoard。
- 复制按钮：对应 Edit 菜单项的 Copy to ClipBoard。
- 粘贴按钮：对应 Edit 菜单项的 Paste from ClipBoard。
- 块复制按钮：复制一个块区域。
- 块移动按钮：移动一个块区域。
- 块旋转按钮：旋转一个块区域。
- 块删除按钮：删除一个块区域。
- 打开元件库按钮：对应 Library 菜单项的 Pick Device/Symbol。
- 生成器件按钮：对应 Library 菜单项的 Make Device。
- 打包工具按钮：对应 Library 菜单项的 Packaging Tool。
- 排序按钮：对应 Library 菜单项中的 Decompose。
- 打开或者关闭自动布线按钮：对应 Tools 菜单项中的 Wire Auto Router。
- 搜索标签项按钮：对应 Tools 菜单项中的 Search and Tag。
- 属性编辑按钮：对应 Tools 菜单项中的 Property Assignment Tool。
- 打开设计浏览器按钮：对应 Design 菜单项中的 Design explore。
- 新建一个文件按钮：对应 Design 菜单项中的 New Sheet。
- 删除文件按钮：对应 Design 菜单项中的 Remove Sheet。



- 返回到当前文件按钮：返回到当前的文件。
- 查看当前设计器件清单按钮：打开当前项目的器件清单。
- 查看当前设计的电气规则检查报表按钮：打开当前项目的电气规则检查报表。
- 转换 ARES 按钮：由当前设计的原理图对应的网络表生成对应的 PCB 图。




## 2. 工具箱

Proteus 的工具箱位于界面的左侧，提供了一些用于图形设计的命令和一些快捷工具箱命令，具体的使用方法将在后续章节中进行进一步介绍，其分类说明如下。

- Selection Mode 按钮：将光标切换到选择模式。
- Component Mode 按钮：将光标切换到元器件操作模式。
- Junction Dot Mode 按钮：将光标切换到节点操作模式。
- Wire Label Mode 按钮：将光标切换到连线标签操作模式。
- Text Script Mode 按钮：将光标切换到文本编辑模式。
- Buses Mode 按钮：将光标切换到总线操作模式。
- Subcircuit Mode 按钮：将光标切换到子电路设计模式。
- Terminals Mode 按钮：将光标切换到终端设计模式，包括输入、输出、电源和地等终端设计。
- Device Pins Mode 按钮：将光标切换到引脚设计模式，在对象选择器中列出各种引脚，如普通引脚、时钟引脚、反电压引脚和短接引脚等。
- Graph Mode 按钮：在对象选择器中列出各种仿真分析所需的图表，如模拟图表、数字图表、混合图表和噪声图表等。
- Tape Recorder Mode 按钮：当需要对设计电路分割仿真时可以使用该按钮。
- Generator Mode 按钮：在对象选择器中列出各种激励源，如正弦激励源、脉冲激励源、指数激励源和 FILE 激励源等。
- Voltage Probe Mode 按钮：在原理设计图中添加电压探针，当进行仿真操作时可显示各探针处的电压值。
- Current Probe Mode 按钮：在原理设计图中添加电流探针，当进行仿真时可显示各探针处的电流值。
- Virtual Instruments Mode 按钮：在对象选择器中列出各种虚拟仪器以供调用，如示波器、逻辑分析仪、定时/计数器和模式发生器等。
- 2D Graphics Line Mode 按钮：进入画线模式。
- 2D Graphics Box Mode 按钮：进入画方块模式。
- 2D Graphics Circle Mode 按钮：进入画圆模式。
- 2D Graphics Arc Mode 按钮：进入圆弧设计模式。
- 2D Graphics Closed Path Mode 按钮：进入封闭区域设计模式。
- 2D Graphics Text Mode 按钮：进入文本编辑模式。
- 2D Graphics Symbols Mode 按钮：进入编辑 2D 符号模式。
- 2D Graphics Maker Mode 按钮：进入 2D 图形标记工具模式。
- Rotate Clockwise 按钮：顺时针方向旋转按钮，以 90° 偏置改变元器件的放置方向。





- Rotate Anti - clockwise 按钮：逆时针方向旋转按钮，以 90°偏置改变元器件的放置方向。
- X - mirror 按钮：水平镜像旋转按钮，以 Y 轴为对称轴，按 180°偏置旋转元器件。
- Y - mirror 按钮：垂直镜像旋转按钮，以 X 轴为对称轴，按 180°偏置旋转元器件。

### 4.3 Proteus 的使用流程

Proteus ISIS 原理图的设计完整流程包括新建设计文档，放置元器件等 8 个步骤，如图 4.41 所示，其详细说明如下。

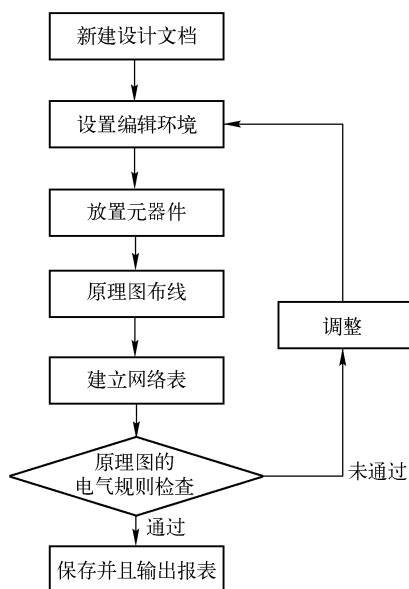


图 4.41 Proteus 设计流程

(1) 新建设计文档。在进入原理图设计之前，首先要构思好原理图，即必须知道所设计的项目需要哪些电路来完成，用何种模板；然后在 Proteus ISIS 编辑环境中画出电路原理图。

(2) 设置工作环境。根据实际电路的复杂程度来设置图纸的大小等。在电路图设计的整个过程中，图纸的大小可以不断地调整。设置合适的图纸大小是完成原理图设计的第一步。

(3) 放置元器件。首先从添加元器件对话框中选取需要添加的元器件，将其布置到图纸的合适位置，并对元器件的名称、标注进行设定；再根据元器件之间的走线等联系对元器件在工作平面上的位置进行调整和修改，使得原理图美观、易懂。

(4) 原理图布线。根据实际电路的需要，利用 Proteus ISIS 编辑环境所提供的各种工具、命令进行布线，将工作平面上的元器件用导线连接起来，构成一幅完整的电路原理图。

(5) 建立网络表。在完成上述步骤之后，即可看到一张完整的电路图，但要完成印制板电路的设计，还需要生成一个网络表文件。网络表是印制板电路与电路原理图之间的纽带。

(6) 原理图的电气规则检查。当完成原理图布线后，利用 Proteus ISIS 编辑环境所提供的电气规则检查命令对设计进行检查，并根据系统提示的错误检查报告修改原理图。

(7) 调整。如果原理图已通过电气规则检查，那么原理图的设计就完成了，但是对于一般电路设计而言，尤其是较大的项目，通常需要对电路进行多次修改才能通过电气规则检查。

(8) 保存并且输出报表。Proteus ISIS 提供了多种报表输出格式，同时可以对设计好的原理图和报表进行保存和输出打印。

### 4.4 Proteus 中的 ATmega16 及其使用

Proteus 中的 ATmega16 位于 Microprocessor ICs 库的 AVR Family 子分类库中，它提供了

大量的各种 AVR 单片机器件，如图 4.42 所示。

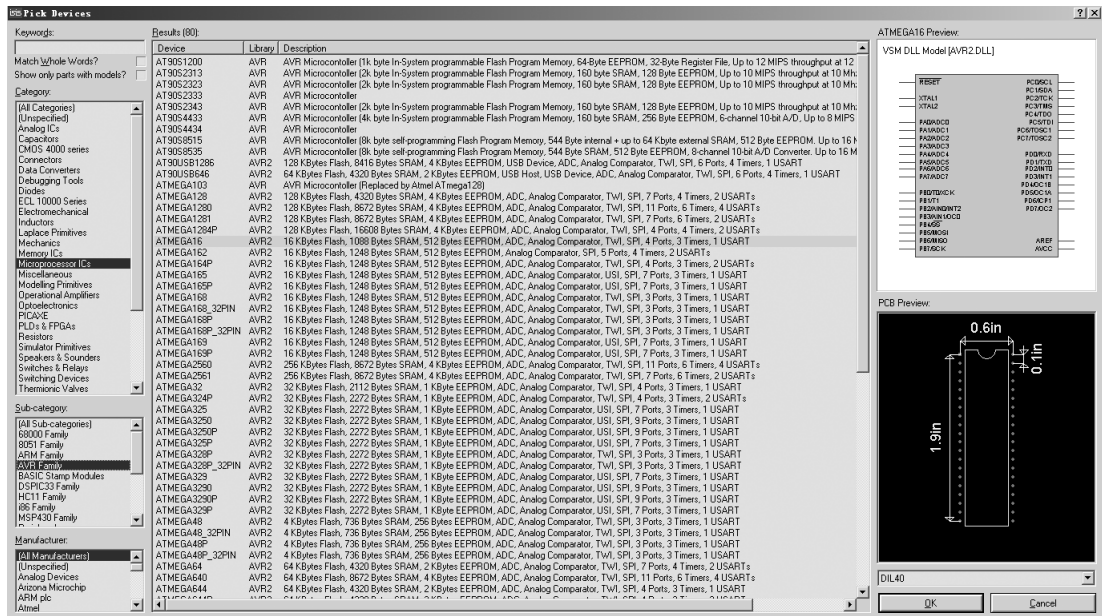


图 4.42 Proteus 中的 ATmega16

在 ATmega16 上双击可以弹出如图 4.43 所示的属性设置对话框，其中涉及的主要参数说明如下。

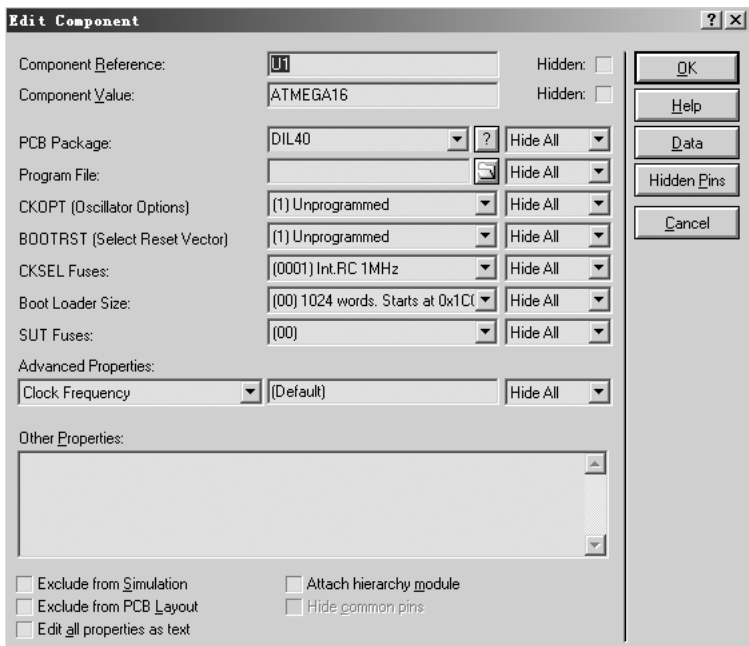


图 4.43 ATmega16 属性设置对话框

- ProgramFile: 编程文件, 在仿真中 ATmega16 所调用的可执行源文件。
- CKOPT: 外部晶体选择控制, 当选择 Programmed 时可以使用高于 8MHz 的外部晶体, 否则只能使用 8MHz 及 8MHz 以下的外部晶体, 内部 RC 振荡源不受该影响, 通常来说如果使用内部的振荡源, 该位使用默认值即可。
- BOOTRST: 决定 ATmega16 复位之后时第一条执行指令的地址。默认状态为 “Unprogrammed”, 表示启动时从 0x0000 开始执行, 否则启动时从 Boot Loader 区的起始地址处开始执行程序。
- CKSEL Fuses: 熔丝设置位, 决定 ATmega128 使用的时钟源, 本书的实例通常会使用 Int RC 8MHz (内部 8MHz RC 时钟源)。
- Boot Loader Size: Boot Loader 区的大小, 通常来说, 使用默认值即可。
- SUT Fuse: 启动时间长度选择, 通常来说, 使用默认值即可。
- Advanced Properties: 高级选项设置, 包括时钟频率、初始化 E<sup>2</sup>PROM 和反编译三个选项, 通常来说, 都使用默认值即可。

## 4.5 Proteus 和 ICC AVR 联合使用

Proteus 作为一个硬件仿真环境, 支持和 ICC AVR 开发环境进行联合调试, 可以对 ATmega16 应用系统进行综合仿真, 本节以一个简单的应用实例来介绍如何进行联合调试。

### 1. 建立仿真电路图和源文件

使用 Proteus 和 ICC AVR 进行联合调试, 首先要建立相应的仿真电路图和 C51 语言的源文件, 其详细操作步骤如下。

(1) 新建一个 Proteus ISIS 电路图文件, 添加如图 4.44 所示的电路, 其涉及的典型器件如表 4.1 所示。

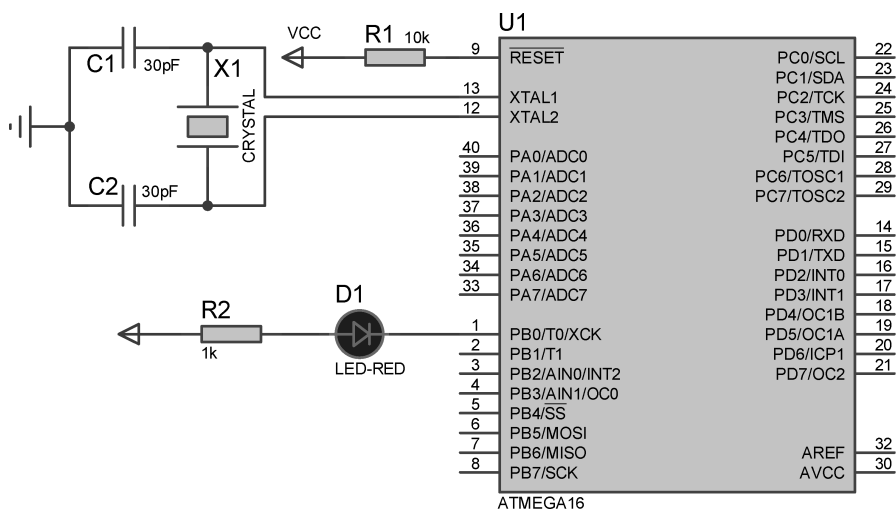


图 4.44 ICC AVR 和 Proteus 联合调试



表 4.1 应用实例器件列表

器件名称	大 类 库	子 类 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
LED - RED	Optoelectronics	LEDs	发光二极管（红色）

(2) 在 ICC AVR 中新建一个命名为“Exam401Union”的工程文件，输入如例 4.1 所示的代码，并且编译生成对应的 .hex 和 .coff 文件。



### 注意

编辑 ICC AVR 工程文件的方法可以参考第 3.7 节，需要注意的是 ICC AVR 不支持中文的项目文件名。

### 【例 4.1】 Proteus 和 ICC AVR 进行联合调试实例代码

```
#include <iom16v.h>
#include <macros.h>

//1ms 延时函数
void delay_1ms(void)
{
    unsigned int i;
    for(i = 1; i < (unsigned int)(8 * 143 - 2); i++) ;
}

//ms 延时函数
void delay_ms(unsigned int time)
{
    unsigned int i = 0;
    while(i < time)
    {
        delay_1ms();
        i++;
    }
}

//初始化端口
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
```



```

PORTB = 0x00;
DDRB  = 0x01;          //PB0 设置为输出
PORTC = 0x00;
DDRC  = 0x00;
PORTD = 0x00;
DDRD  = 0x00;
}
//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();

    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主函数
void main( void)
{
    init_devices();
    while(1)
    {
        PORTB = 0x01;      //PB0 输出高电平
        delay_ms(50);      //延迟 50ms
        PORTB = 0x00;      //PB0 输出低电平
        delay_ms(50);      //延迟 50ms
    }
}

```

(3) 双击 Proteus 电路中的 ATmega16，弹出如图 4.43 所示属性设置对话框，在 Program File 中选择上一步中生成的 .cof 文件，并且参考图 4.43 设置好 ATmega16 的其他属性。

(4) 点击运行，可以看到发光二极管闪烁，如图 4.45 所示。

## 2. 联合调试中仿真的运行控制

仿真的运行控制包括“Start/Restart Debugging”（启动仿真）、“Pause Amiation”（暂停仿真）、“Stop Amiation”（停止仿真），同时也可以使用快捷工具栏中的相应按键控制仿真，如图 4.46 所示，这四个按钮功能分别是：启动仿真、单步运行、暂停仿真、停止仿真。



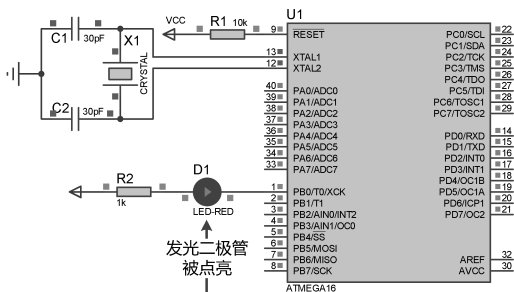


图 4.45 实例仿真运行结果



图 4.46 快捷工具栏中的仿真运行控制

3. 联合调试中的仿真数据记录

双击仿真运行控制快捷工具栏右侧的“Messages”按钮可以调出如图 4.47 的记录窗口，其中记录了仿真中的一些相关信息，如果在仿真中出现错误或者警告，也会在其中体现。

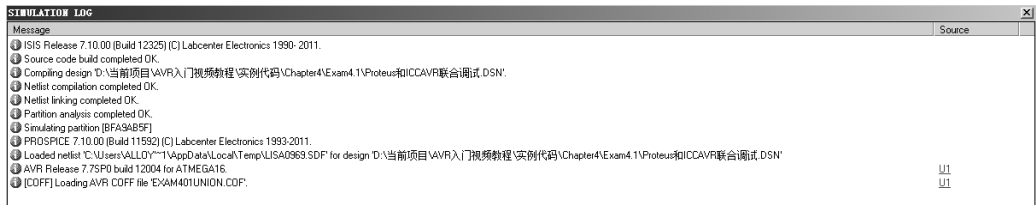


图 4.47 仿真数据记录窗口

4. ATmega16 的内部资源观测窗口

点击暂停按钮，打开 Debug 菜单，可以看到如图 4.48 所示的 ATmega16 单片机内部资源观测窗口，其详细说明如下。

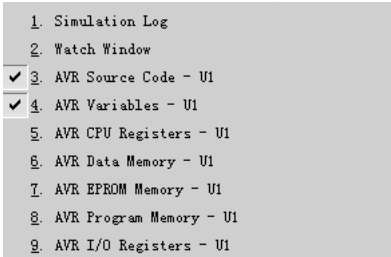



图 4.48 ATmega16 单片机内部资源观测窗口



注意

在设置 ATmega128 属性时，选择的执行文件必须为 .cof 文件，否则将不会出现 AVR Source Code 和 AVR Variables 窗口。

- AVR Source Code: ATmega16 的源代码窗口，如图 4.49 所示，在其中可以进行相应的调试操作，如单步、断点等。

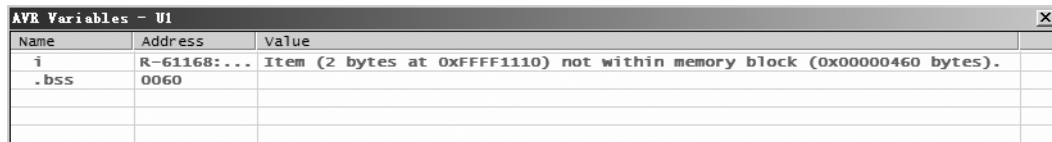


```

---- //ICC-AVR application builder : 2015/4/8 20:29:34
---- // Target : M16
---- // Crystal: 8.0000Mhz
----
---- #include <iom16v.h>
---- #include <macros.h>
----
---- //1ms延时函数
---- void delay_1ms(void)
---- {
----     unsigned int i;
009A for(i=1;i<(unsigned int)(8*143-2);i++) ;
---- }
---- //ms延时函数
---- void delay_ms(unsigned int time)
---- {
00B6     unsigned int i=0;
00C2     while(i<time)
----     {
00BC         delay_1ms();
00BE         i++;
----     }
---- }
---- //初始化端口
---- void port_init(void)
---- {
00CC     PORTA = 0x00;
00D0     DDRA = 0x00;
00D2     PORTB = 0x00;
00D4     DDRB = 0x01; //PB0设置为输出
00D8     PORTC = 0x00;
00DA     DDRC = 0x00;
00DC     PORTD = 0x00;
00DE     DDRD = 0x00;
---- }
---- //初始化ATmega16
---- void init_devices(void)
    
```

图 4.49 AVR Source Code 窗口

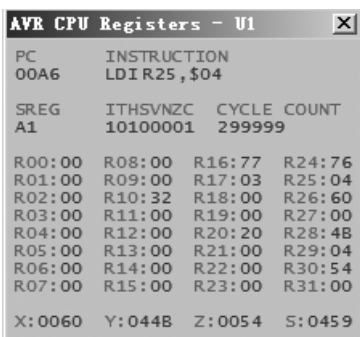
- AVR Variables: ATmega16 的代码变量窗口，用于跟踪调试，观察相应变量的数据，如图 4.50 所示。



Name	Address	Value
i	R-61168:...	Item (2 bytes at 0xFFFF1110) not within memory block (0x00000460 bytes).
.bss	0060	

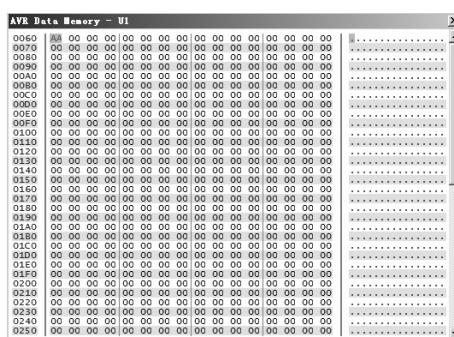
图 4.50 AVR Variables 窗口

- AVR CPU Registers: ATmega16 的内部寄存器观察窗口，用于显示仿真过程中 ATmega16 的内部寄存器数据状态，如图 4.51 所示。
- AVR Data Memory: ATmega16 的内部数据存储器观察窗口，用于显示仿真过程中 ATmega16 内部数据存储器数据状态，如图 4.52 所示。



AVR CPU Registers - U1			
PC	INSTRUCTION		
00A6	LDI R25, \$04		
SREG	ITHSVNZC CYCLE COUNT		
A1	10100001 299999		
R00:00	R08:00	R16:77	R24:76
R01:00	R09:00	R17:03	R25:04
R02:00	R10:32	R18:00	R26:60
R03:00	R11:00	R19:00	R27:00
R04:00	R12:00	R20:20	R28:4B
R05:00	R13:00	R21:00	R29:04
R06:00	R14:00	R22:00	R30:54
R07:00	R15:00	R23:00	R31:00
X:0060	Y:044B	Z:0054	S:0459

图 4.51 AVR CPU Registers 窗口



AVR Data Memory - U1	
0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0110	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0200	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0210	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0220	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0230	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0240	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0250	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 4.52 AVR Data Memory 窗口

- AVR EPROM Memory: ATmega128 的内部 E<sup>2</sup>PROM 寄存器的观察窗口, 用于显示 ATmega128 的内部 E<sup>2</sup>PROM 的数据状态, 如图 4.53 所示。
- AVR Program Memory: ATmega16 的程序存储器观察窗口, 可以看到当前用户代码对应的机器码存放状态, 如图 4.54 所示。

AVR EPROM Memory - U1			
0000	FF	FF	FF
0008	FF	FF	FF
0010	FF	FF	FF
0018	FF	FF	FF
0020	FF	FF	FF
0028	FF	FF	FF
0030	FF	FF	FF
0038	FF	FF	FF
0040	FF	FF	FF
0048	FF	FF	FF
0050	FF	FF	FF
0058	FF	FF	FF
0060	FF	FF	FF
0068	FF	FF	FF
0070	FF	FF	FF
0078	FF	FF	FF
0080	FF	FF	FF
0088	FF	FF	FF
0090	FF	FF	FF
0098	FF	FF	FF
00A0	FF	FF	FF
00A8	FF	FF	FF
00B0	FF	FF	FF
00B8	FF	FF	FF
00C0	FF	FF	FF
00C8	FF	FF	FF
00D0	FF	FF	FF
00D8	FF	FF	FF
00E0	FF	FF	FF
00E8	FF	FF	FF
00F0	FF	FF	FF
00F8	FF	FF	FF

图 4.53 AVR EPROM Memroy 窗口

AVR Program Memory - U1			
0000	0C	94	2A
0010	FF	FF	FF
0020	FF	FF	FF
0030	FF	FF	FF
0040	FF	FF	FF
0050	FF	FF	FF
0060	0A	EA	08
0070	11	F0	01
0080	10	E0	00
0090	0D	92	F9
00A0	0F	5F	1F
00B0	0E	94	87
00C0	5F	4F	4A
00D0	2A	BA	28
00E0	08	95	F8
00F0	08	95	F7
0100	22	24	28
0110	4A	93	BA
0120	08	95	FF
0130	FF	FF	FF
0140	FF	FF	FF
0150	FF	FF	FF
0160	FF	FF	FF
0170	FF	FF	FF
0180	FF	FF	FF
0190	FF	FF	FF
01A0	FF	FF	FF
01B0	FF	FF	FF
01C0	FF	FF	FF
01D0	FF	FF	FF
01E0	FF	FF	FF
01F0	FF	FF	FF

图 4.54 AVR Program Memory 窗口

- AVR I/O Registers: ATmega128 的 I/O 寄存器观察窗口, 用于显示当前 ATmega128 的 I/O 状态, 如图 4.55 所示。

AVR I/O Registers - U1			
20	00	00	00
30	00	00	00
40	06	00	00
50	00	80	00

图 4.55 AVR I/O Registers 窗口

# 第5章 ATmega16 单片机的 I/O 引脚和外部中断

I/O 引脚是 ATmega16 和外部进行数据信号交互的通道，本章介绍 ATmega16 的通用 I/O 引脚和外部中断的基础知识与它们的应用方法，以及发光二极管（LED）、按键和键盘的使用方法。

## 5.1 ATmega16 外部引脚基础使用方法

ATmega16 单片机有 32 个通用 I/O 引脚，分为 PA ~ PD 4 组，每组包括 Px0 ~ Px7 共 8 个引脚，这些引脚都有对应的控制寄存器、数据寄存器和方向寄存器，并且也都有第二功能。

### 5.1.1 ATmega16 的 I/O 引脚的结构

ATmega16 的单个 I/O 引脚的内部结构如图 5.1 所示，每个引脚的输出缓冲器都具有对称的驱动能力，可以输出或吸收大电流，支持直接驱动某些外部器件如发光二极管等；所有的端口引脚内部都具有钳位电容以及与电压无关的上拉电阻，并有保护二极管与电源（ $V_{CC}$ ）和地信号相连。

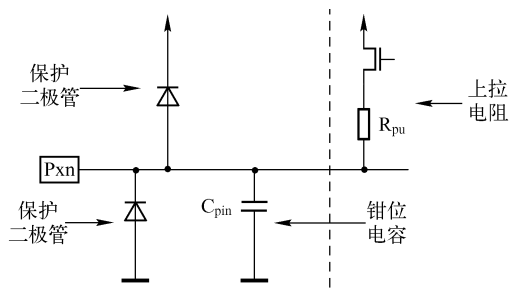


图 5.1 ATmega16 的 I/O 引脚结构

ATmega16 的每个 I/O 端口（PA ~ PD）都有如下 3 个 I/O 寄存器地址。

- (1) 数据寄存器 PORTx（x = A ~ D）：控制 ATmega16 对应引脚的输出电平。
- (2) 数据方向寄存器 DDRx：控制 ATmega16 对应引脚的输入、输出方向。
- (3) 端口输入引脚寄存器 PINx：存放 ATmega16 对应引脚的输入电平。

其中数据寄存器和数据方向寄存器为读/写寄存器，而端口输入引脚寄存器为只读寄存器。

**注意**

对端口输入引脚寄存器 PIN<sub>x</sub> 某一位写入逻辑“1”，将造成数据寄存器相应位的数据发生“0”与“1”的交替变化，并且当寄存器 SFI/OR 的上拉禁止位 PUD 置位时所有端口引脚的上拉电阻都被禁止。

### 5.1.2 ATmega16 的 I/O 引脚配置

ATmega16 的每个端口引脚都有 Px0 ~ Px7 共 8 个位，每个位都对应 DDR<sub>x</sub>、PORT<sub>x</sub>、PIN<sub>x</sub> 3 个寄存器中的对应位 DD<sub>xn</sub>、PORT<sub>xn</sub> 和 PIN<sub>xn</sub>，如表 5.1 至表 5.12 所示，其中“R/W”表示该寄存器位既能进行读操作，也能进行写操作，“R”表示该寄存器位只能进行读操作；初始值为“0”表示上电复位之后该寄存器的位为低电平，“N/A”表示该寄存器位的数值不可预测。

**表 5.1 ATmega16 的端口 A 数据寄存器 PORTA**

BIT	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

**表 5.2 ATmega16 的端口 A 方向寄存器 DDRA**

BIT	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

**表 5.3 ATmega16 的端口 A 输入引脚寄存器 PINA**

BIT	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
读/写	R	R	R	R	R	R	R	R
初始值	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

**表 5.4 ATmega16 的端口 B 数据寄存器 PORTB**

BIT	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

**表 5.5 ATmega16 的端口 B 方向寄存器 DDRB**

BIT	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0



表 5.6 ATmega16 的端口 B 输入引脚寄存器 PINB

BIT	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
读/写	R	R	R	R	R	R	R	R
初始值	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

表 5.7 ATmega16 的端口 C 数据寄存器 PORTC

BIT	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

表 5.8 ATmega16 的端口 C 方向寄存器 DDRC

BIT	DDRC7	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

表 5.9 ATmega16 的端口 C 输入引脚寄存器 PINC

BIT	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
读/写	R	R	R	R	R	R	R	R
初始值	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

表 5.10 ATmega16 的端口 D 数据寄存器 PORTD

BIT	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

表 5.11 ATmega16 的端口 D 方向寄存器 DDRD

BIT	DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

表 5.12 ATmega16 的端口 D 输入引脚寄存器 PIND

BIT	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
读/写	R	R	R	R	R	R	R	R
初始值	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

ATmega16 的 DDR 寄存器中的 DDR<sub>xn</sub> 位用来选择引脚的方向，当 DDR<sub>xn</sub> 为“1”时，引脚 P<sub>xn</sub> 被配置为输出状态，否则配置为输入状态。当引脚配置为输入状态时，若 PORT<sub>xn</sub> 为“1”上拉电阻将使能。如果需要关闭这个上拉电阻，可以将 PORT<sub>xn</sub> 清零，或者将这个引脚配置为输出，ATmega16 上电复位时各引脚为高阻态。当引脚配置为输出状态时，若 PORT<sub>xn</sub> 为“1”，对应的引脚 P<sub>xn</sub> 输出高电平，否则输出低电平。

需要注意的是，当 ATmega16 的引脚在高阻态（{DDR<sub>xn</sub>, PORT<sub>xn</sub>} = 0b00）和输出高电平

( $\{DDx_n, PORTx_n\} = 0b11$ ) 两种状态之间进行切换时, 上拉电阻使能 ( $\{DDx_n, PORTx_n\} = 0b01$ ) 或输出低电平 ( $\{DDx_n, PORTx_n\} = 0b10$ )。这两种模式必然会有一个发生, 通常来说, 上拉电阻被使能是完全可以接受的, 因为高阻状态不用在意是高电平输出还是上拉输出, 若和实际使用情况不符, 则可以通过置位 SFI/OR 寄存器的 PUD 位来禁止所有端口的上拉电阻。在上拉输入和输出低电平之间切换也有同样的问题。用户必须选择高阻态 ( $\{DDx_n, PORTx_n\} = 0b00$ ) 或输出高电平 ( $\{DDx_n, PORTx_n\} = 0b11$ ) 作为中间步骤。

表 5.13 为 ATmega16 的端口引脚配置总结表。

表 5.13 ATmega16 的端口引脚配置表

DDX <sub>n</sub>	PORTX <sub>n</sub>	PUD (IN SFI/OR)	I/O	上拉电阻	说 明
0	0	X	输入	无	高阻态
0	1	0	输入	有	被外部电路拉低时将输出电流
0	1	1	输入	无	高阻
1	0	X	输出	无	输出低电平 (漏电流)
1	1	X	输出	无	输出低电平 (源电流)

### 5.1.3 ATmega16 的 I/O 引脚电平读取

不管当前 ATmega16 的 DDR<sub>x</sub> 寄存器被配置为输入状态还是输出状态, 都可以通过读取 PINX<sub>n</sub> 寄存器来获得对应的引脚电平, 其对应的读取时序如图 5.2 所示。可以看到, 其中最大和最小传输延迟分别为  $t_{pd, max}$  和  $t_{pd, min}$ , 从系统时钟第一个下降沿之后的起始时钟周期可以看到, 当时钟信号为低时锁存器是关闭的; 而时钟信号为高时信号可以自由通过, 如图中 SYNCLATCH 信号的阴影区所示。当时钟为低时信号即被锁存, 然后在紧接着的系统时钟上升沿锁存到 PINX<sub>n</sub> 寄存器, 如  $t_{pd, max}$  和  $t_{pd, min}$  所示, 引脚上的信号转换延迟在  $\frac{1}{2} \sim 1\frac{1}{2}$  个系统时钟之间。

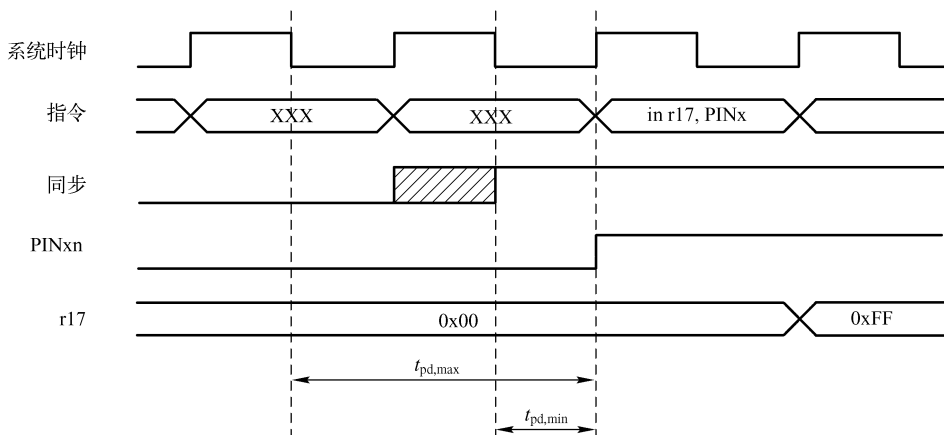


图 5.2 ATmega16 读取引脚电平时同步器的时序

所以当在 ATmega16 的 I/O 引脚上写入一个电平之后需要读出这个引脚的输入状态, 则



需要在赋值指令和读取指令之间有一个时钟周期的间隔。

PINxn 寄存器的各个位与其前面的锁存器组成了一个同步器，这样可以避免在内部时钟。状态发生改变的短时间范围内由于引脚电平变化而造成的信号不稳定，其缺点是引入了延迟。

#### 5.1.4 ATmega16 的 I/O 引脚低功耗处理

ATmega16 的 I/O 引脚可以由休眠控制寄存器在各种掉电模式、省电模式以及 Standby 模式下通过设置 SLEEP 信号进入低功耗模式，以防止在输入悬空或模拟输入电平接近  $V_{CC}/2$  时消耗太多的电流。

当引脚作为外部中断输入时 SLEEP 信号无效，但若外部中断没有使能，SLEEP 信号仍然有效，如果引脚的第二功能使能的时候 SLEEP 也让位于第二功能（参考 5.2 节）。



##### 注意

如果逻辑高电平出现在一个被设置为“上升沿、下降沿或任何逻辑电平变化都引起中断”的外部异步中断引脚上，即使该外部中断未被使能，但从上述休眠模式唤醒时，相应的外部中断标志位仍会被置“1”。这是因为引脚电平在休眠模式下被钳位到低电平，而唤醒过程造成了引脚电平从“0”到“1”的变化。

如果 ATmega16 有引脚未被使用，建议赋予这些引脚一个确定的电平，虽然在深层休眠模式下大多数数字输入被禁用，但还是需要避免因引脚没有确定的电平而造成悬空引脚在其他数字输入使能模式（复位、工作模式、空闲模式）消耗电流。

最简单的保证未用引脚具有确定电平的方法是使能内部上拉电阻。但要注意的是，复位时上拉电阻将被禁用。如果复位时的功耗也有严格要求，则建议使用外部上拉或下拉电阻，不推荐直接将未用引脚与  $V_{CC}$  或 GND 信号连接，因为这样可能会在引脚偶然作为输出时出现冲击电流。

## 5.2 ATmega16 外部引脚的第二功能

ATmega16 的 I/O 引脚除了可以作为一般的数字 I/O 端口使用之外，都有自己对应的第二功能，本节简要介绍这些 I/O 引脚的第二功能，它们的具体使用方法参考后面的对应章节。

### 1. PORTA 引脚的第二功能

ATmega16 的端口 A 的第二功能是 ADC 模块的模拟信号输入通道，其说明如表 5.14 所示。需要注意的是，如果 PORTA 的引脚被配置为输入，则在进行 AD 转换时不能切换输出/输入状态，否则会影响 AD 转换的结果。

表 5.14 PORTA 的第二功能

端 口 引 脚	第 二 功 能
PA7	ADC7（ADC 输入通道 7）



端 口 引 脚	第 二 功 能
PA6	ADC6 (ADC 输入通道 6)
PA5	ADC5 (ADC 输入通道 5)
PA4	ADC4 (ADC 输入通道 4)
PA3	ADC3 (ADC 输入通道 3)
PA2	ADC2 (ADC 输入通道 2)
PA1	ADC1 (ADC 输入通道 1)
PA0	ADC9 (ADC 输入通道 0)

## 2. PORTB 引脚的第二功能

ATmega16 的端口 B 的第二功能主要包括 SPI 总线模块接口、模拟比较器信号输入和定时计数器的外部信号输入, 如表 5.15 所示, 其中部分引脚可能有两个可选项, 其详细说明如下。

表 5.15 PORTB 的第二功能

端 口 引 脚	第 二 功 能
PB7	SCK (SPI 总线的串行时钟)
PB6	MISO (SPI 总线的主机输入/从机输出信号)
PB5	MOSI (SPI 总线的主机输出/从机输入信号)
PB4	SS (SIP 从机选择引脚)
PB3	AIN1 (模拟比较负信号输入) OCO (T/C0 输出比较匹配输出)
PB2	AIN0 (模拟比较正信号输入) INT2 (外部中断 2 输入)
PB1	T1 (T/C1 外部计数器输入)
PB0	T0 (T/C0 外部计数器输入) XCK (USART 外部时钟输入/输出)

- SCK: SPI 总线接口的主机时钟输出、从机时钟输入端口。当 ATmega16 工作于从机模式时, 不论 DDB7 配置为什么方向, 该引脚都将被设置为输入; 当 ATmega16 工作于主机模式时, 该引脚的数据方向由 DDB7 控制, 当被设置为输入时其内部上拉电阻由 PORTB7 控制。
- MISO: SPI 总线接口的主机数据输入、从机数据输出端口。当 ATmega16 工作于主机模式时, 不论 DDB6 配置为什么方向, 该引脚都将被设置为输入; 当 ATmega16 工作于从机模式时, 该引脚的数据方向由 DDB6 控制, 当设置为输入时其内部上拉电阻由 PORTB6 控制。
- MOSI: SPI 总线接口的主机数据输出、从机数据输入端口。当 ATmega16 工作于从机模式时, 不论 DDB5 配置为什么方向, 该引脚都将被设置为输入; 当 ATmega16 工作于主机模式时, 该引脚的数据方向由 DDB5 控制, 当设置为输入后其内部上拉电阻由



PORTB5 控制。

- SS：SPI 总线接口的从机选择输入。当 ATmega16 工作于从机模式时，不论 DDB4 配置为什么方向，该引脚都将设置为输入，当此引脚为低时 SPI 被激活；当 ATmega16 工作于主机模式时，该引脚的数据方向由 DDB4 控制，当被设置为输入时其内部上拉电阻由 PORTB4 控制。
- AIN1（OC0）：模拟比较负输入（AIN1）。当配置该引脚为输入时，应该切断内部上拉电阻，以防止数字端口功能与模拟比较器功能相冲突。该引脚还可以作为输出比较匹配输出（OC0），可以作为 T/C0 比较匹配的外部输出，此时该引脚必须配置为输出，在 PWM 模式的定时功能中，OC0 引脚作为输出。
- AIN0（INT2）：模拟比较正输入（AIN0）。当配置该引脚为输入时，应该切断内部上拉电阻以防止数字端口功能与模拟比较器功能相冲突。此外，该引脚还可以作为外部中断源 2（INT2）的输入引脚。
- T1：T/C1 计数器源输入。
- T0（XCK）：可以作为 T/C0 计数器源输入（T0）；还可以作为串口（USART）的外部时钟输入（XCK），其中数据方向寄存器的第 0 位（DDB0）用于控制时钟为输出（DDB0 置位）还是输入（DDB0 清零），需要注意的是，只有当串口工作在同步模式时，该引脚才被激活。

### 3. PORTC 引脚的第二功能

ATmega16 的端口 C 的第二功能如表 5.16 所示，它主要用于定时振荡器引脚、JTAG 接口引脚和 TWI（I<sup>2</sup>C）总线接口引脚，其详细说明如下。

表 5.16 PORTC 的第二功能

端 口 引 脚	第 二 功 能
PC7	TOSC2（定时振荡器引脚 2）
PC6	TOSC1（定时振荡器引脚 1）
PC5	TDI（JTAG 测试数据输入）
PC4	TDO（JTAG 测试数据输出）
PC3	TMS（JTAG 测试模式选择）
PC2	TCK（JTAG 测试时钟）
PC1	SDA（两线串行总线接口数据线）
PC0	SCL（两线串行总线接口时钟线）

- TOSC2：定时振荡器引脚 2。当寄存器 ASSR 的 AS2 位被置“1”，使能 T/C2 的异步时钟时引脚 PC7 将端口断开，成为振荡器放大器的反向输出。在这种模式下，外部晶体振荡器与该引脚相连，该引脚不能作为 I/O 引脚。
- TOSC1：定时振荡器引脚 1。当寄存器 ASSR 的 AS2 位置“1”，使能 T/C2 的异步时钟时，引脚 PC6 将端口断开，成为振荡器放大器的反向输出。在这种模式下，外部晶体振荡器与该引脚相连，该引脚不能作为 I/O 引脚。
- TDI：JTAG 测试数据输入引脚，用于将串行输入的数据移入指令寄存器或数据寄存



器。当 JTAG 接口被使能时，该引脚不能作为普通 I/O 引脚。

- **TDO**: JTAG 测试数据输出引脚，用于将串行输入数据移出指令寄存器或数据寄存器。当 JTAG 接口被使能时，该引脚不能作为普通 I/O 引脚，它在除 TAP 状态情况外均为三态进入移出数据状态。
- **TMS**: JTAG 测试模式选择引脚，其作为 TAP 控制器状态工具的定位。当 JTAG 接口被使能时，该引脚不能作为普通 I/O 引脚。
- **TCK**: JTAG 测试时钟引脚驱动 JTAG 接口工作。当 JTAG 接口被使能时，该引脚不能作为普通 I/O 引脚。
- **SDA**: 两线串行总线接口 (TWI/I2C) 的数据引脚。当寄存器 TWCR 的 TWEN 位置“1”时使能，该引脚 PC1 不与端口相连，成为两线串行接口的串行数据 I/O 引脚，但在该模式下仍然可以使用 PORTC 来控制上拉电阻。
- **SCL**: 两线串行总线接口 (TWI/I2C) 的时钟引脚。当 TWCR 寄存器的 TWEN 位置“1”时，使能两线串行接口，引脚 PC0 不与端口相连，成为两线串行接口的串行时钟 I/O 引脚，但在该模式下仍然可以使用 PORTC 来控制上拉电阻。

#### 4. PORTD 引脚的第二功能

ATmega16 的端口 D 的第二功能如表 5.17 所示，它主要用于定时计数器的输入捕捉、输出比较、外部中断输入和串口 (USART) 的输入/输出引脚，其详细说明如下。

表 5.17 PORTC 的第二功能

端 口 引 脚	第 二 功 能
PD7	OC2 (T/C2 输出比较匹配输出)
PD6	ICP1 (T/C1 输入捕捉引脚)
PD5	OC1A (T/C2 输出比较 A 匹配输出)
PD4	OC1B (T/C1 输出比较 B 匹配输出)
PD3	INT1 (外部中断 1 输入)
PD2	INT0 (外部中断 0 输入)
PD1	TXD (USART 输出引脚)
PD0	RXD (USART 输入引脚)

- **OC2**: 定时计数器 T/C2 输出比较外部输入引脚，而当 T/C2 工作于 PWM 模式下时该引脚作为输出。
- **ICP1**: 定时计数器 T/C1 的输入捕捉引脚。
- **OC1A**: 定时计数器 T/C2 比较匹配 A 输出的外部输入引脚，而当 T/C2 工作于 PWM 模式下时，该引脚作为输出。
- **OC1B**: 定时计数器 T/C1 比较匹配 B 输出的外部输入引脚，而当 T/C1 工作于 PWM 模式下时，该引脚作为输出。
- **INT1**: 外部中断 1 输入引脚。
- **INT0**: 外部中断 0 输入引脚。
- **TXD**: 串口 (USART) 模块的数据发送引脚。当使能 USART 的发送器时，该引脚被



强制设置为输出，此时 DDR 寄存器不起作用。

- RXD：串口（USART）模块的数据接收引脚。当使能 USART 的接收器时，该引脚被强制设置为输入，此时 DDR 寄存器不起作用，但是 PORTD 仍然可以对该引脚的上拉电阻进行控制。

## 5.3 ATmega16 的外部中断

ATmega16 提供了 INT0、INT1 和 INT2 三个外部中断源，它们可以用于监视外部的中断事件以供单片机及时做出相应的处理。其中 INT0 和 INT1 可以设置为上升沿、下降沿以及低电平触发方式，而 INT2 可以设置为上升沿或者下降沿触发方式。

当 ATmega16 的外部中断控制寄存器使能后，在三个外部中断源对应的引脚上出现对应的电平或边沿的变化时，ATmega16 会检测到中断事件，即使这些引脚被配置为输出状态。



### 注意

如果 INT0 和 INT1 配置低电平触发模式，只要中断使能，当外部引脚上加上低电平时，将产生中断事件；但如果将其配置为沿触发模式，则必须在系统时钟正常工作的前提下才会产生中断事件。

ATmega16 通过对相应的寄存器操作来对外部中断进行控制，这些寄存器包括 MCU 控制寄存器（MCUCR）、MCU 控制与状态寄存器（MCUCSR）、通用中断控制寄存器（GICR）和通用中断标志寄存器（GIFR）。

### 5.3.1 MCU 控制寄存器（MCUCR）

ATmega16 的 MCU 控制寄存器包含中断触发控制位与通用 MCU 功能，其内部结构如表 5.18 所示。

表 5.18 ATmega16 的 MCU 控制寄存器 MCUCR

BIT	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

MCUCR 寄存器的 ISC11 和 ISC10 位用于设置外部中断 1 的中断触发方式。如果 SREG 寄存器的 I 标志位和相应的中断屏蔽位被置位的话，外部中断 1 由外部 I/O 引脚 INT1 上的信号触发，其触发方式如表 5.19 所示，在检测边沿前 ATmega16 首先采样 INT1 引脚上的电平信号，如果选择了边沿触发方式或电平变化触发方式，那么持续时间大于一个时钟周期的脉冲将触发中断，过短的脉冲则不能保证触发中断；如果选择低电平触发方式，那么低电平必须保持到当前指令执行完成。

表 5.19 外部中断 1 的触发方式

ISC11	ISC10	触发方式
0	0	低电平触发方式

ISC11	ISC10	触发方式
0	1	任意逻辑电平变化都会引发中断
1	0	下降沿触发中断
1	1	上升沿触发中断

MCUCR 寄存器的 ISC01 和 ISC00 位用于设置外部中断 0 的中断触发方式。如果 SREG 寄存器的 I 标志位和相应的中断屏蔽位被置位的话, 外部中断 0 由外部 I/O 引脚 INT0 上的信号触发, 其触发方式如表 5.20 所示, 在检测边沿前 ATmega16 首先采样 INT0 引脚上的电平信号, 如果选择了边沿触发方式或电平变化触发方式, 那么持续时间大于一个时钟周期的脉冲将触发中断, 过短的脉冲则不能保证触发中断; 如果选择低电平触发方式, 那么低电平必须保持到当前指令执行完成。

表 5.20 外部中断 0 的触发方式

ISC01	ISC0	触发方式
0	0	低电平触发方式
0	1	任意逻辑电平变化都会引发中断
1	0	下降沿触发中断
1	1	上升沿触发中断

### 5.3.2 MCU 控制与状态寄存器 (MCUCSR)

ATmega16 的 MCU 控制与状态寄存器 MCUCSR 中用于选择外部中断 INT2 的触发方式, 其内部结构如表 5.21 所示。

表 5.21 ATmega16 的 MCU 控制与状态寄存器 MCUCSR

BIT	JTD	ISC2	—	JTRF	WDRF	BORF	EXTFR	PORF
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	见具体说明				

MCUCSR 寄存器和外部中断相关的位是 ISC2, 这是中断 2 触发方式的控制位。如果 SREG 寄存器的 I 标志和 GICR 寄存器相应的中断屏蔽位被置位的话, 外部中断 2 由外部 I/O 引脚 INT2 上的信号触发。若 ISC2 置“0”, 则 INT2 引脚上加载的下降沿触发中断; 若 ISC2 置“1”, 则 INT2 引脚加载的上升沿触发中断。INT2 的边沿触发方式是异步的, 只要 INT2 引脚上产生宽度大于 50ns 的脉冲就会引发中断。若选择了低电平中断, 低电平必须保持到当前指令完成, 然后才会产生中断, 而且只要将引脚拉低, 就会触发中断, 改变 ISC2 时有可能发生中断, 因此建议首先在寄存器 GICR 里清除相应的中断使能位 INT2, 然后再改变 ISC2。

### 5.3.3 通用中断控制寄存器 (GICR)

通用中断控制寄存器 GICR 用于对 ATmega16 的中断事件进行管理, 其内部结构结果如



表 5.22 所示。当 INT1、INT0 和 INT2 对应的位被置“1”且状态寄存器 SREG 的 I 标志被置“1”时，允许对应的外部中断引脚上的中断触发信号，当对应位被清零时，禁止对应外部中断引脚上的中断触发信号。

表 5.22 ATmega16 的通用中断控制寄存器 GICR

BIT	INT1	INT0	INT2	—	—	—	IVSEL	IVCE
读/写	R/W	R/W	R/W	R	R	R	R/W	R/W
初始值	0	0	0	0	0	0	0	0

### 5.3.4 通用中断标志寄存器（GIFR）

ATmega16 的通用中断标志寄存器 GIFR 用于记录外部中断的状态，其内部结构如表 5.23 所示。当外部中断引脚触发了中断事件之后对应寄存器位将被置位，ATmega16 的程序指针将自动跳转到相应的中断向量，当进入中断服务程序之后该标志位会自动被清零，另外也可以通过对这些位写入“1”来清除中断标志。

表 5.23 ATmega16 的通用中断标志寄存器 GIFR

BIT	INTF1	INTF0	INTF2	—	—	—	—	—
读/写	R/W	R/W	R/W	R	R	R	R	R
初始值	0	0	0	0	0	0	0	0

需要注意的是，当 INT0 被设置为低电平触发方式时，INTF0 会被自动清零，并且当 INT2 中断禁用进入某些休眠模式时，该引脚的输入缓冲将禁用，这会导致 INTF2 标志设置信号的逻辑变化。

## 5.4 ATmega16 的 I/O 引脚和中断的应用实例

本节提供一些 ATmega16 的 I/O 引脚和中断的应用实例，这些实例都有实例配合讲解以供读者理解。

### 5.4.1 I/O 引脚输出高低脉冲电平实例

本应用是一个使用 ATmega16 的 PORTB 端口输出高低脉冲电平的实例。

#### 1. 实例的设计思路

设置 PORTB 端口为输出端口，使用软件延时之后在对应的端口上轮流输出高电平和低电平即可制造出脉冲效果。

#### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 5.3 所示，一个虚拟示波器连接到 ATmega16 的 PB0 引脚上，表 5.24 给出了实例中使用的 Proteus 器件。

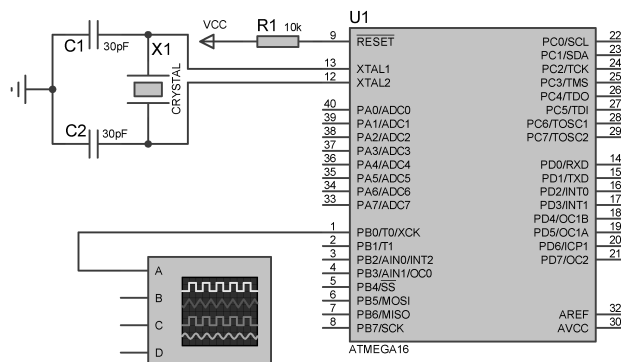


图 5.3 实例的 Proteus 电路

表 5.24 应用实例器件列表

器件名称	大类库	子类库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体

### 3. 实例的应用代码

实例的应用代码如例 5.1 所示。

应用代码首先对 ATmega16 的 I/O 引脚的功能进行初始化，通过设定 DDRB 来确定 PORTB 引脚为输出状态，然后使用两个 for 循环语句来进行延时，在第一个延时中，将 PORTB 上的电平置为高电平，在另外一个延时中将 PORTB 上的电平置为低电平。

#### 【例 5.1】 I/O 引脚输出高低脉冲电平

```
#include <iom16v.h>
#include <macros.h>
```

```
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0xFF;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
```



```

}
void init_devices(void)
{
    CLI();
    port_init();

    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}
void main(void)
{
    unsigned int i;           //软件计数器
    init_devices();
    while(1)
    {
        for(i = 0; i < 50000; i ++ );    //软件延时
        PORTB = 0xFF;                  //PA 端口输出高电平
        for(i = 0; i < 25000; i ++ );    //软件延时, 略短
        PORTB = 0x00;                  //PA 端口输出低电平
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，可以看到在 ATmega16 的 PORTB 引脚上电平发生了变化，调节示波器可以看到对应的波形，如图 5.4 所示。

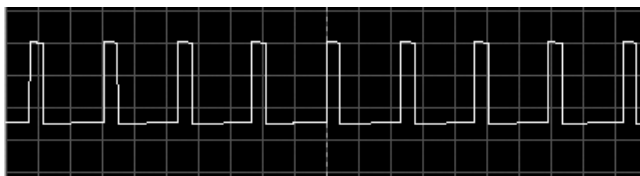


图 5.4 实例的仿真运行结果



#### 总结

调节示波器，可以看到在输出波形中低电平的宽度是高电平宽度的近似 2 倍，这是因为低电平的 for 循环延时时间是高电平的 2 倍，读者可以自行尝试通过调整相应的延时次数来调整其宽度。

#### 5. Proteus 中的虚拟示波器

虚拟示波器（OSCILLOSCOPE）是用来观察当前电路某个点的波形变化的仪器，是 Pro-



teus 仿真中最常用的虚拟仪器。

在 Proteus 中,单击工具箱中的“Virtual Instrument Mode”按钮图标,此时当前窗口会出现包括虚拟示波器的所有虚拟仪器的列表,在该列表中选择虚拟示波器后在电路图中点击,即可放置虚拟示波器,如图 5.5 所示。

在仿真运行下,点击 Debug 菜单下的 Digital OSCILLOSCOPE 选项可以打开虚拟示波器的窗口,可以分为波形输出区域、触发 (Trigger) 设置区域、水平 (Horizontal) 设置区域、通道 (Channel A ~ Channel D) 设置区域,如图 5.6 所示。

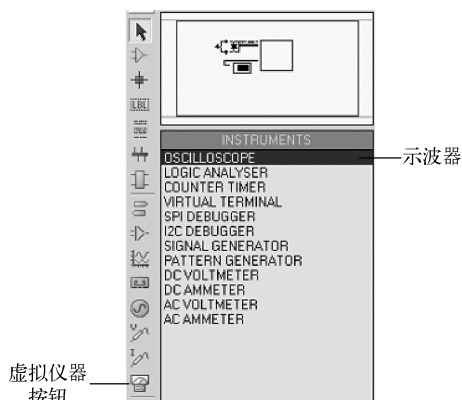


图 5.5 虚拟仪器列表中的示波器

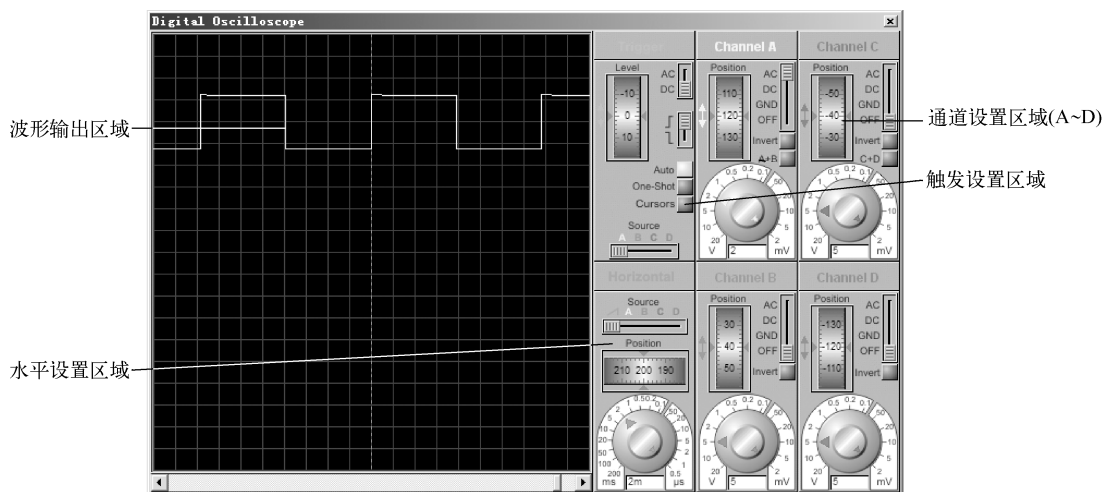



图 5.6 虚拟示波器的输出区域


波形输出区域用于显示待输出波形,可以显示单通道到最多 4 个通道的输出波形。

触发设置区域的相关按键、波轮、开关等用于设置所有通道信号的触发方式,包括水平参考线位置设置 (Level)、触发信号交直流设置 (AC、DC)、触发方式设置、光标开关、触发源设置。

- 水平参考线位置设置 (Level): 用于设置当前水平参考线的位置,通过拖动波轮可以使得其在  $-210 \sim 210$  之间移动,同时波形输出区域的水平参考线将上下移动。
- 触发信号交直流设置: 用于设置待监视信号是交流信号还是直流信号,有“AC”和“DC”两个开关选项, .
- 触发方式设置: 当设置为“Auto”时,随着输入信号的刷新,虚拟示波器的输出波形会自动跟随刷新;当设置为“One - Shot”时,则只捕捉一帧波形,然后保持。
- 光标 (Cursors) 开关: 当该按钮被按下时会在虚拟示波器的波形输出区域跟随鼠标状态放置一些参考线显示对应点的电压、时间信息等,并且可以拖动。

- 触发源设置：将虚拟示波器的触发源分别设置为 Channel A ~ Channel D，在完成设置之后，水平参考线会出现在当前选择通道上。

水平设置区域的相关按键、波轮、开关等用于设置所有通道信号输出的水平显示参数，包括参考源（Source）设置、位置设置（Position）和显示时间刻度设置。

- 参考源设置：用于设置在显示区域中显示的波形的相对参考位置，包括水平、A、B、C、D 五个不同选项，通常来说选择水平即可，.
- 位置设置：用于控制显示波形左右移动。
- 显示时间刻度设置：用于调整波形输出区域一个刻度所代表的时间长度，取值范围为 0.5 ~ 200ms，下方的大箭头用于整刻度调整，上面的小刻度用于小刻度调整。

通道设置区域用于设置各个通道的相关参数，分为 Channel A ~ Channel D，它们是完全相同的，包括位置设置（Position）、触发设置、特殊功能开关、电压刻度设置。

- 位置设置：用于设置该通道在波形输出显示区域的位置，该波轮与水平参考线位置设置波轮使用方法类似，通过调节该波轮可以将对应的输出信号（A ~ D）在显示输出区域中移动。
- 触发设置：单独设置该通道的触发信号，包括交流（AC）、直流（DC）、地（GND）、关闭（OFF）。
- 特殊功能开关：包括 Invert（翻转）和重叠，前者将对应的波形翻转，后者将波形重叠，只有 Channel A 和 Channel C 具有该开关，分别对应“A + B”和“C + D”。



#### 注意

在使用“A + B”或者“C + D”功能时，对应的 Channel B 或者 Channel D 的波形就不会输出了。

- 电压刻度设置：与显示时间刻度设置类似，该波轮用于修改调整波形输出区域一个刻度所代表的电压宽度，取值范围为 20V 到 2mV（毫伏），同样是大、小箭头配合使用。

### 5.4.2 I/O 引脚驱动发光二极管（LED）实例

本实例是一个用发光二极管实现的“流水灯”应用，8 个发光二极管循环轮流点亮，实现流水效果。

#### 1. 发光二极管（LED）基础

发光二极管 LED 和普通二极管一样，具有单向导电性，当加在发光二极管两端的电压超过了它的导通电压（一般为 1.7 ~ 1.9V）时就会导通，当流过它的电流超过一定的电流（一般为 2 ~ 3mA）时则会发光。

ATmega16 应用系统中发光二极管的典型应用电路可以分为“灌电流”和“拉电流”两种，如图 5.7 所示。

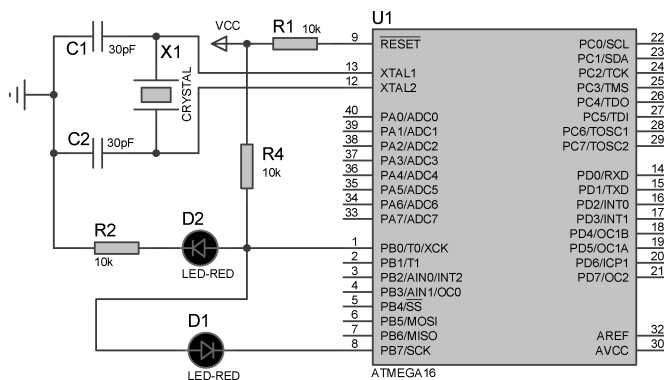


图 5.7 发光二极管的典型应用电路

图 5.7 中 PB7 引脚上的发光二极管 D1 驱动方式称为“灌电流”驱动方式，当 PB7 输出高点平时，D1 两端无电压差，不发光；当 PB7 输出低电平时，D1 两端有 5V 电压差，发光。图 5.7 中 PB0 引脚上的发光二极管 D2 的驱动方式为“拉电流”驱动方式，当 PB0 输出高点平时，D2 相对地有 5V 的电压差，发光；当 PB0 输出低电平时，5V 电压差将落在上拉电阻 R4 上，D2 两端无电压差，不发光。图 5.7 中的 R2 和 R3 都是限流电阻，当电阻值较小时，电流较大，发光二极管亮度较高；当电阻值较大时，电流较小，发光二极管亮度较低。

## 2. Proteus 中的发光二极管

Proteus 中的发光二极管位于 Optoelectronics 库的 LEDs 子分类库中，并提供了 DIODE - LED、LED、LED - RED 等多种发光二极管元件，如图 5.8 所示。

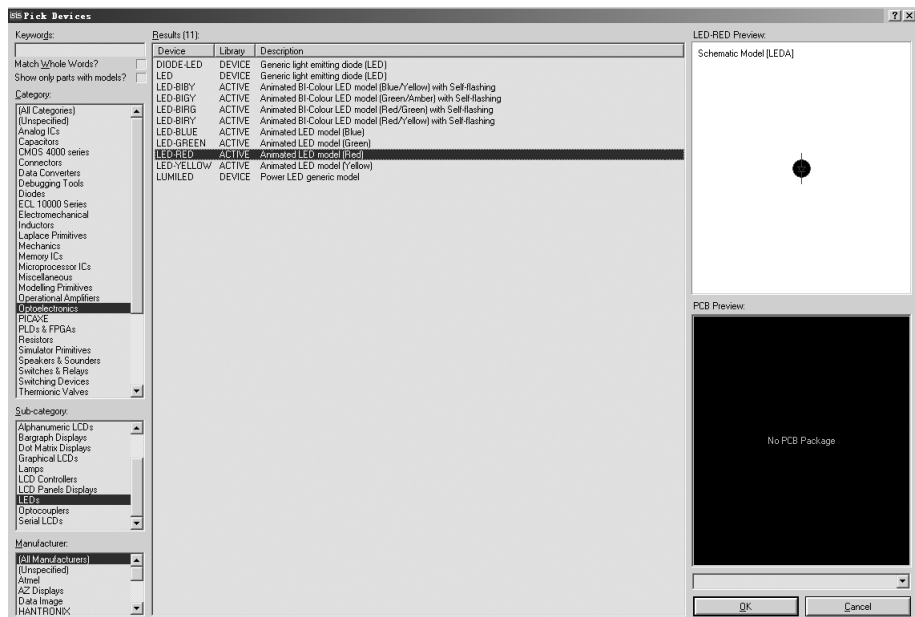


图 5.8 Proteus 中的发光二极管

通常来说，选择 LED - RED/LED - BLUE/LED - GREEN 即可，其分别对应红色、蓝色和绿色的发光二极管。



### 注意

要想在仿真中看到发光二极管的实际运行情况，必须选择 ACTIVE 类型的发光二极管，而不能选择 DEVICE 类型的。

在发光二极管上双击可以弹出如图 5.9 所示的属性设置对话框，其中涉及的主要参数说明如下。

- Forward Voltage: 导通电压，当该发光二极管导通时其两端的电压差。
- Full drive current: 最大驱动电流。

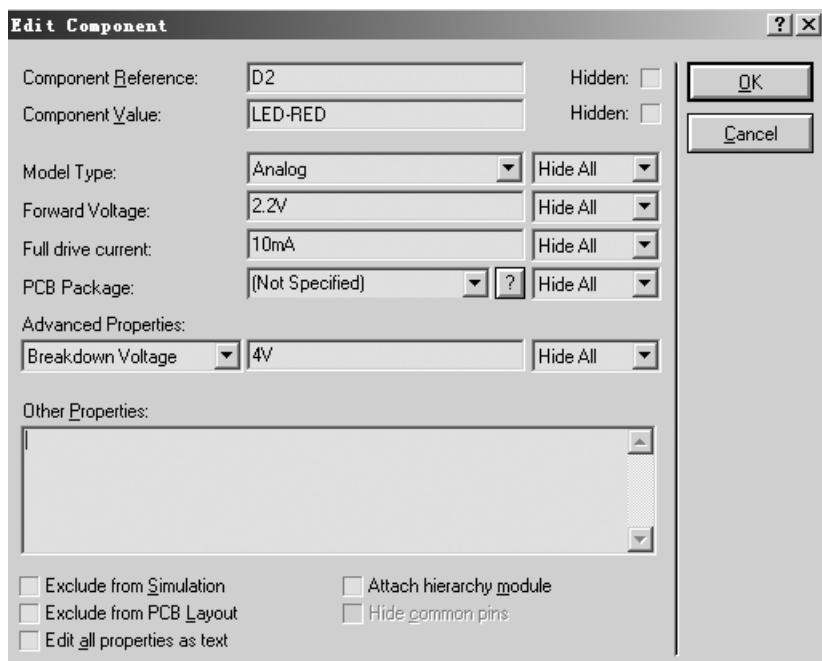


图 5.9 发光二极管的属性设置对话框

### 3. 实例设计思路

为了实现流水效果，可以让 ATmega16 从其 I/O 端口上轮流输出对应的电平，在灌电流驱动方式下，输出高电平可以关闭发光二极管，输出低电平可以点亮发光二极管，在两次状态之间添加一定的延时，即可达到“流水”的效果。

### 4. 实例的 Proteus 电路图

实例的 Proteus 电路如图 5.10 所示，ATmega16 使用 PB 端口“灌电流”驱动方式驱动了 8 个发光二极管 D1 ~ D8，使用一个电阻排作为限流电阻，电路中的发光二极管除了标号都使用默认设置，表 5.25 为实例中使用的 Proteus 器件。

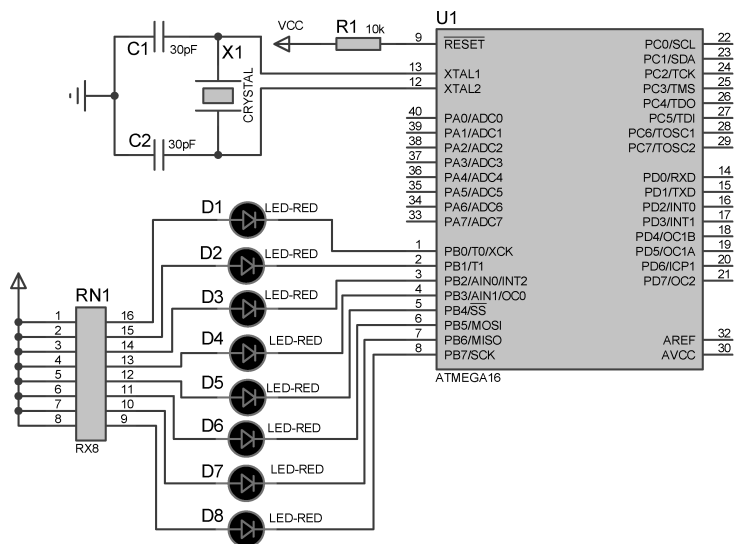


图 5.10 实例的 Proteus 电路

表 5.25 Proteus 电路器件列表

器件名称	库	子 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16
RES	Resistors	Generic	通用电阻
RX8	Resistors	Resistors Packs	8 位电阻排
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
LED - RED	Optoelectronics	LEDs	发光二极管 (红色)

## 5. 实例的应用代码

实例的应用代码如例 5.2 所示。

使用一个 unsigned char 类型的变量 LED 来存放待输出的发光二极管状态，先将该变量初始化为 0x01，然后在一个延时间隔之后对该变量进行移位操作后将该变量通过 P1 端口送出，由于硬件电路使用了“灌电流”的驱动方式，所以要对该变量进行取反后再送出，这样可以保证同一个时刻只有一个发光二极管被点亮。

代码使用函数 void DelayMs (unsigned int time) 来进行软件延时，修改参数 time 的长度可以修改“流水”的速度。

### 【例 5.2】“流水灯”

```
#include <iom16v.h>
#include <macros.h>

void delay(void)           //延时函数
{
```

```

    unsigned int i;
    for(i = 1; i < 100; i ++ );
}

void delay_1ms( void)           //1ms 延时函数
{
    unsigned int i;
    for(i = 1; i < ( unsigned int)( 8 * 143 - 2); i ++ );
}

void DelayMs( unsigned int time)    //ms 延时函数
{
    unsigned int i = 0;
    while( i < time)
    {
        delay_1ms( );
        i ++ ;
    }
}

void port_init( void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0xFF;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
}

void init_devices( void)
{
    CLI( );
    port_init( );

    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI( );
}

void main( void)
{
    unsigned char LED = 0x01;

```

```

init_devices();
while(1)
{
    DelayMs(100);           //延时
    if(LED == 0x80)
        //判断是否到最高位,如果流水到头,则折返到最低位点亮
    {
        LED = 0x01;
    }
    else
    {
        LED = LED << 1;    //移位,行程流水
    }
    PORTB = ~LED;
}

```

## 6. 实例的仿真结果和说明

点击运行, 可以看到 D1 ~ D8 依次轮流点亮, 如图 5.11 所示。

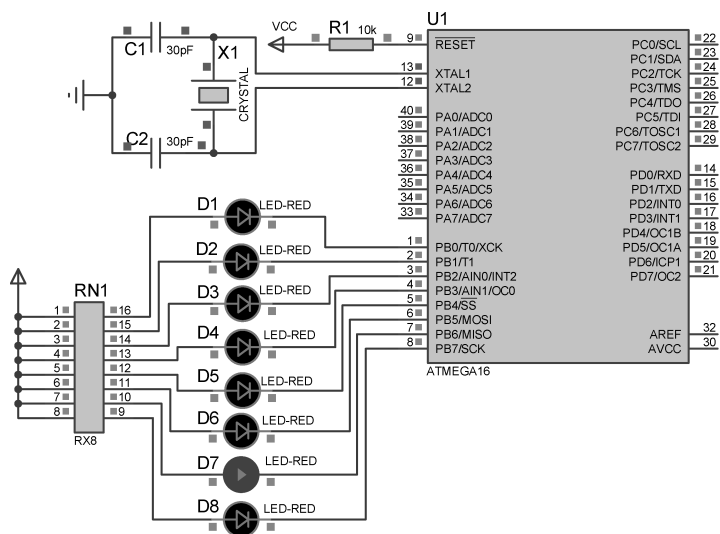


图 5.11 实例的仿真运行结果



## 总结

读者可以自行修改变量 LED 的循环方式。例如, 每次输出两个低电平点亮两个发光二极管, 或者间隔点亮多个发光二极管; 还可以修改延时时长, 来营造不同的“流水”效果。

### 5.4.3 I/O 引脚驱动单位数码管实例

发光二极管只能用于显示比较简单的状态，当 ATmega16 需要显示数字、字符等信息时，可以使用数码管。本实例是一个用单位数码管实现的“流水数字显示”应用，单位数码管轮流显示“0～9”的数字。

#### 1. 单位数码管基础

数码管是一种由多个发光二极管组成的半导体发光器件，常见的数码管可以按照显示的段数分为七段数码管、八段数码管和异型数码管；按能显示多少个字符/数字可以分为一位、两位等“X”位数码管；按照数码管中各个发光二极管的连接方式可以分为共阴极数码管和共阳极数码管。

数码管的本质是组合在一起的 8 个发光二极管，通过点亮不同的发光二极管组合可用来显示数字 0～9、字符 a、f、h、l、p、r、u、y、符号“-”及小数点“.”，图 5.12 是数码管的引脚定义和内部等效发光二极管结构，可以看到共阴极数码管和共阳极数码管的内部连接方式是不同的。

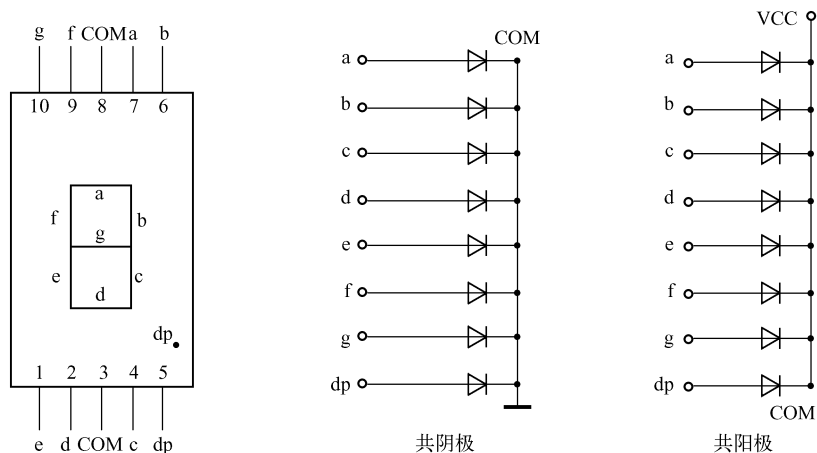


图 5.12 数码管的内部结构

从图 5.12 可以看到，当数码管内部的发光二极管被点亮时，对应的数码管段发光，所以可以根据数码管需要显示的数字或者字符推导出需要外加在数码管引脚上的电平组合，这个过程称为对数码管进行字形编码，由于共阴极和共阳极的数码管结构不同，所以对应的编码也不同，如表 5.26 所示。

表 5.26 八段数码管的字形编码

显示 字符	共阳极数码管									共阴极数码管								
	dp	g	f	e	d	c	b	a	代码	dp	g	f	e	d	c	b	a	代码
0	1	1	0	0	0	0	0	0	C0H	0	0	1	1	1	1	1	1	3FH
1	1	1	1	1	1	0	0	1	F9H	0	0	0	0	0	1	1	0	06H
2	1	0	1	0	0	1	0	0	A4H	0	1	0	1	1	0	1	1	5BH



续表

显示 字符	共阳极数码管									共阴极数码管								
	dp	g	f	e	d	c	b	a	代码	dp	g	f	e	d	c	b	a	代码
3	1	0	1	1	0	0	0	0	B0H	0	1	0	0	1	1	1	1	4FH
4	1	0	0	1	1	0	0	1	99H	0	1	1	0	0	1	1	0	66H
5	1	0	0	1	0	0	1	0	92H	0	1	1	0	1	1	0	1	6DH
6	1	0	0	0	0	0	1	0	82H	0	1	1	1	1	1	0	1	7DH
7	1	1	1	1	1	0	0	0	F8H	0	0	0	0	0	1	1	1	07H
8	1	0	0	0	0	0	0	0	80H	0	1	1	1	1	1	1	1	7FH
9	1	0	0	1	0	0	0	0	90H	0	1	1	0	1	1	1	1	6FH
a	1	0	0	0	1	0	0	0	88H	0	1	1	1	0	1	1	1	77H
b	1	0	0	0	0	0	1	1	83H	0	1	1	1	1	1	0	0	7CH
c	1	1	0	0	0	1	1	0	C6H	0	0	1	1	1	0	0	1	39H
d	1	0	1	0	0	0	0	1	A1H	0	1	0	1	1	1	1	0	5EH
e	1	0	0	0	0	1	1	0	86H	0	1	1	1	1	0	0	1	79H
f	1	0	0	0	1	1	1	0	8EH	0	1	1	1	0	0	0	1	71H
h	1	0	0	0	1	0	0	1	89H	0	1	1	1	0	1	1	0	76H
l	1	1	0	0	0	1	1	1	C7H	0	0	1	1	1	0	0	0	38H
p	1	0	0	0	1	1	0	0	8CH	0	1	1	1	0	0	1	1	73H
r	1	1	0	0	1	1	1	0	CEH	0	0	1	1	0	0	0	1	31H
u	1	1	0	0	0	0	0	1	C1H	0	0	1	1	1	1	1	0	3EH
y	1	0	0	1	0	0	0	1	91H	0	1	1	0	1	1	1	0	6EH
—	1	0	1	1	1	1	1	1	BFH	0	1	0	0	0	0	0	0	40H
.	0	1	1	1	1	1	1	1	7FH	1	0	0	0	0	0	0	0	80H
无	1	1	1	1	1	1	1	1	FFH	0	0	0	0	0	0	0	0	00H

和发光二极管类似, 数码管也有“灌电流”和“拉电流”两种不同的驱动方式, 可以参考 5.4.2 节。

## 2. Proteus 中的单位数码管

Proteus 中的数码管位于 Optoelectronics 库的 7 - Segment Displays 子分类库中, 提供了 7SEG - BCD、7SEG - COM - AN - BLUE、7SEG - COM - CAT - GRN 等多种单位数码管元件, 如图 5.13 所示。

通常来说, 选择 7SEG - COM - ANODE/7SEG - COM - CATHODE 即可, 它们分别对应了共阳极和共阴极的 7 数码管。



### 注意

7SEG - COM - ANODE/7SEG - COM - CATHODE 系列的数码管都是不带小数点的, 如果需要使用带小数点的数码管, 可以选择 7SEG - MPX1 - CA/7SEG - MPX1 - CC。

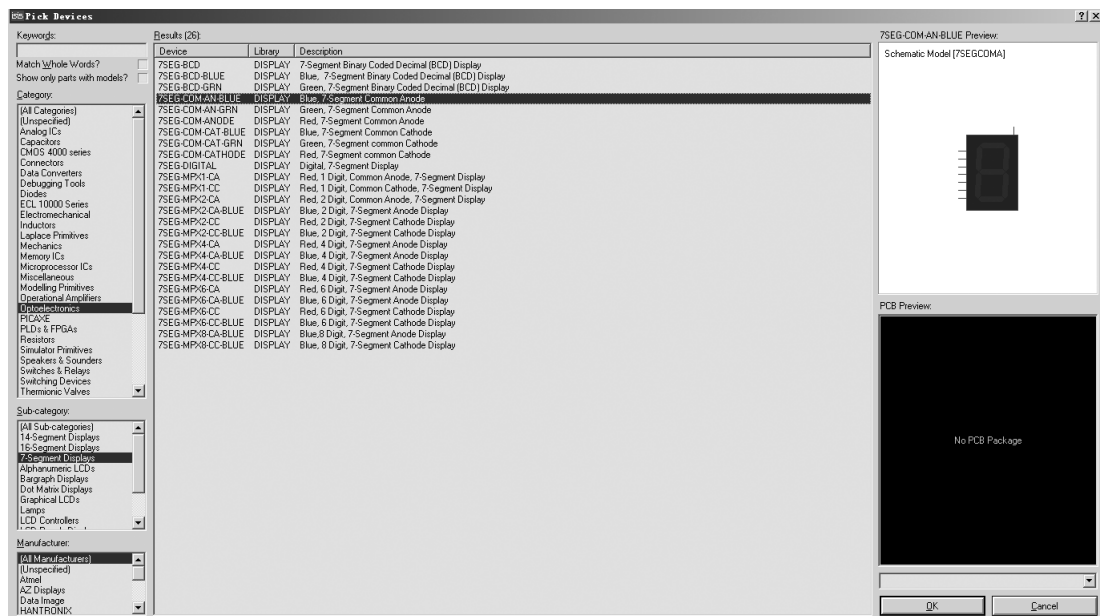


图 5.13 Proteus 中的单位数码管

在单位数码管上双击可以弹出如图 5.14 所示的属性设置对话框，其中涉及的主要参数说明如下。

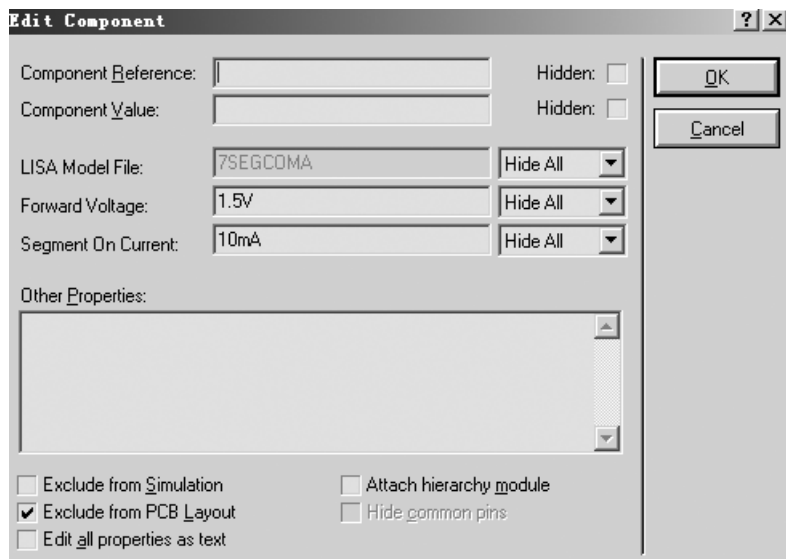


图 5.14 单位数码管的属性设置对话框

- Forward Voltage: 导通电压，当数码管显示时，内部数码管两端对应的电压差值。
- Segment On Current: 段电流，数码管每一段显示时，该段对应的电流，此时数码管需要的总电流大小是点亮的段电流大小之和。

### 3. 实例设计思路

本应用实例设计思路和使用发光二极管构造“流水灯”类似，其差异是依次从单片机端口输出的是需要显示的数字字形编码。

### 4. 实例的 Proteus 电路图

实例的 Proteus 电路如图 5.15 所示，ATmega16 使用 PORTB 端口“灌电流”驱动方式驱动了一个单位数码管 7SEG - COM - ANODE，电路中单位数码管除了标号都使用默认设置，表 5.27 为实例中使用的 Proteus 器件。

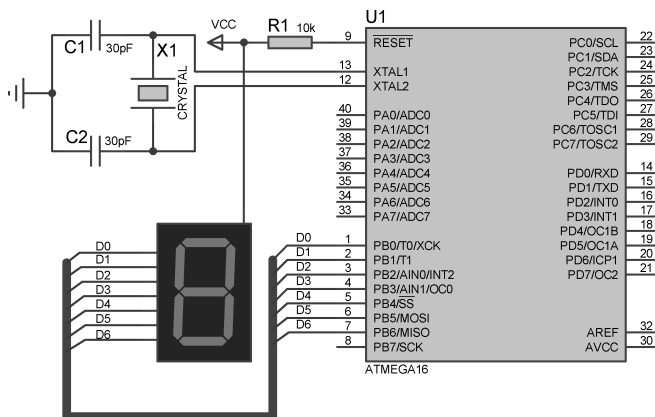


图 5.15 实例的 Proteus 电路



### 注意

数码管同样有限流的问题，当电流过大时可能导致 ATmega16 的 I/O 引脚烧毁，所以在实际使用中，数码管的数据引脚上应参考发光二极管的电路，添加上限流电阻。

表 5.27 Proteus 电路器件列表

器件名称	库	子库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
7SEG - COM - ANODE	Optoelectronics	7 - Segment Displays	七段共阳极数码管

### 5. 实例的应用代码

实例的应用代码如例 5.3 所示。

参考例 5.2 的设计，使用函数 void DelayMs (unsigned int time) 来完成延时操作，然后使用一个 unsigned char 类型的软件计数器变量 counter 来对当前的循环次数进行计数，当计数器值达到 9 时从 0 开始重新计数。应用代码把共阳极数码管对应的字形编码按照对应



“0 ~ 9”的次序存放在 SEGtable 数组中，并且使用 counter 作为查找变量对其进行查找后送到 PORTB 端口驱动数码管显示。

**【例 5.3】“流水”数字显示**

```
#include <iom16v.h>
#include <macros.h>

unsigned char const SEGtable[ ] = {0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90};
// "0" ~ "F" 对应的字形编码, 共阳极八段数码管

void delay( void)                // 延时函数
{
    unsigned int i;
    for( i = 1; i < 100; i ++ );
}

void delay_1ms( void)            // 1ms 延时函数
{
    unsigned int i;
    for( i = 1; i < ( unsigned int )( 8 * 143 - 2 ); i ++ );
}

void DelayMs( unsigned int time) // ms 延时函数
{
    unsigned int i = 0;
    while( i < time )
    {
        delay_1ms( );
        i ++ ;
    }
}

void port_init( void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0xFF;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
}

void init_devices( void)
```

```

}
CLI();
port_init();

MCUCR = 0x00;
GICR  = 0x00;
TIMSK = 0x00;
SEI();
}

void main(void)
{
    unsigned char counter = 0;
    init_devices();
    while(1)
    {
        for( counter = 0; counter < 10; counter ++ )
        {
            DelayMs( 100 );           //延时
            PORTB = SEGtable[ counter ];
        }
    }
}

```

## 6. 实例的仿真结果和说明

点击运行，可以看到数码管循环输出数字“0～9”，如图 5.16 所示。

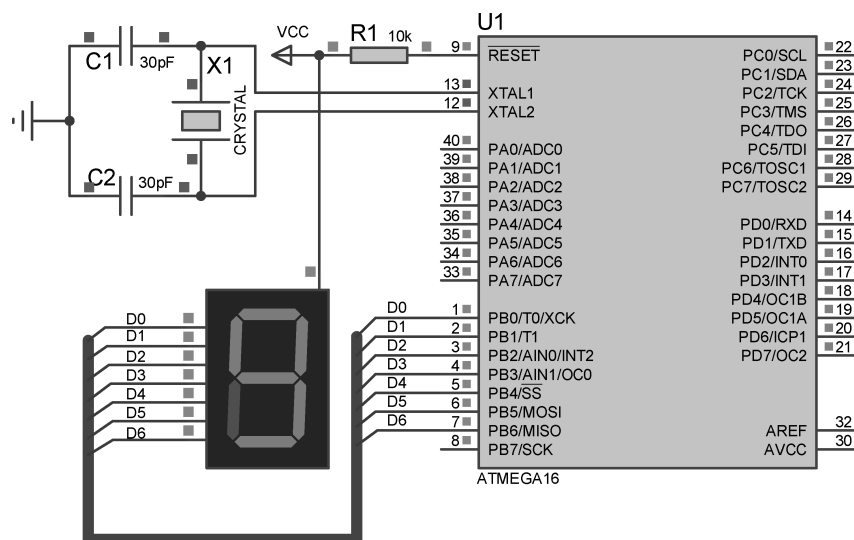


图 5.16 实例的仿真运行结果

## 总结

读者可以自行修改 SEGtable 字符数组中的存储值以将“0 ~ 9”数字修改为“A、B、C、F”等字母进行显示；还可以尝试将数码管修改为 7SEG - COM - CATHODE（共阴极），修改电路驱动方式，并且修改软件代码中的字形编码，然后运行仿真。

### 5.4.4 I/O 引脚驱动独立按键实例

和发光二极管类似，独立按键也是 ATmega16 应用系统中最常用的人机交互通道器件之一，它通常用于给用户提供向 ATmega16 输入信息的通道。本实例是一个用两个独立按键控制数码管实现加减计数的应用，当一个按键被按下一次时，数码管上的显示数字加“1”，当另外一个按键被按下一次时，数码管上的显示数字减“1”。

#### 1. 独立按键基础

独立按键的工作基本原理是，它被按下时，按键接通两个点，放开时则断开这两个点。按照结构可以把按键分为两类：触点式开关按键，如机械式开关、导电橡胶式开关等；无触点开关按键，如电气式按键、磁感应按键等。

ATmega16 应用系统中典型的独立按键应用电路如图 5.17 所示，按键的一端连接到电源地，而另外一端通过一个电阻连接到电源正电压端，同时还连接到 ATmega16 的 I/O 引脚上。当按键没有被按下时，ATmega16 的 I/O 引脚通过电阻连接到  $V_{CC}$  上，I/O 引脚上被加上了一个高电平；当按键被按下时，ATmega16 的 I/O 引脚直接连接到电源地，被加上低电平。

## 注意

图 5.17 中 PB7 引脚上没有加上拉电阻是因为 ATmega16 的 I/O 引脚内置了一个上拉电阻。

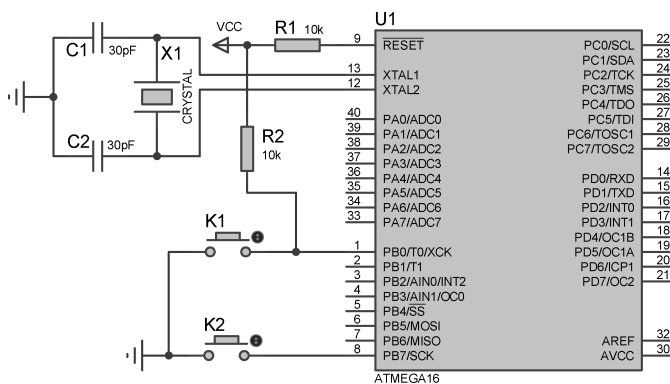


图 5.17 独立按键的典型应用电路

#### 2. Proteus 中的独立按键

Proteus 中的独立按键位于 Switches & Relays 库的 Switches 子分类库中，提供了 BUT-

TON、DIPSSW\_2、JUMPER 等多种开关元件，如图 5.18 所示。

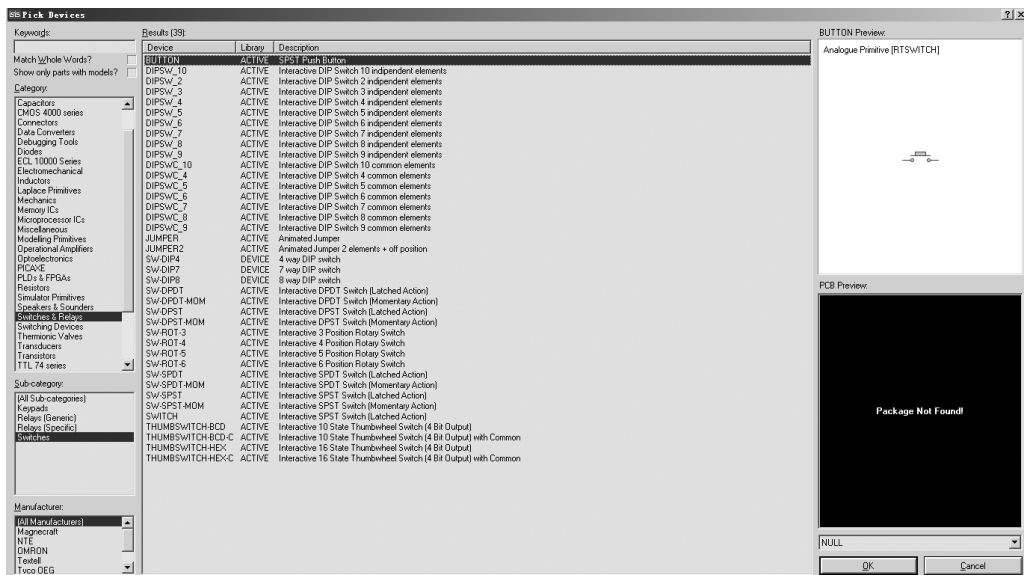


图 5.18 Proteus 中的独立按键

在独立按键上双击可以弹出如图 5.19 所示的属性设置对话框，其中涉及的主要参数说明如下。

- Off Resistance: 当按键断开时两个端点之间的电阻。
- On Resistance: 当按键接通时两个端点之间的电阻。
- Switching Time: 按键断开和接通之间的最短切换时间。

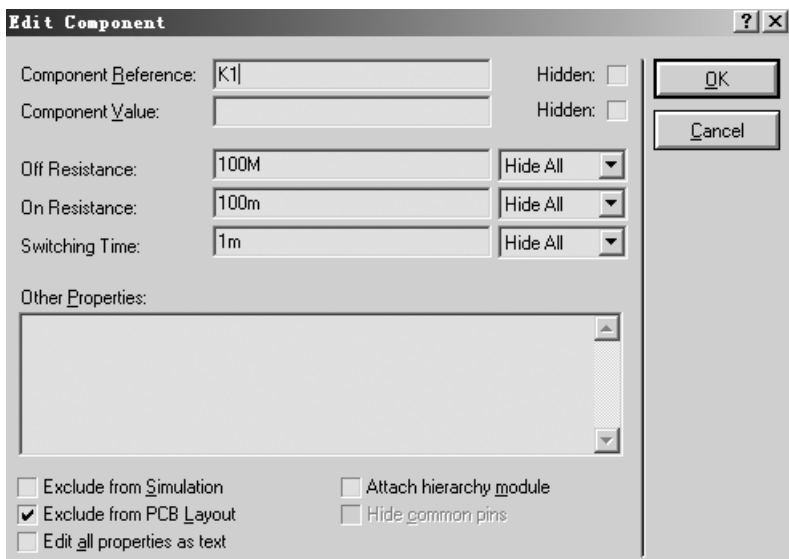


图 5.19 独立按键的属性设置对话框

### 3. 实例设计思路

为了实现计数功能，可以在 ATmega16 内部设置一个软件计数器，当有按键被按下时，对该计数器进行相应的操作，使用该计数器作为参数把其对应的字符编码通过 I/O 口送出驱动数码管显示。

### 4. 实例的 Proteus 电路图

实例的 Proteus 电路如图 5.20 所示，ATmega16 使用 PC3 和 PC7 引脚驱动了 K1 和 K2 两个独立按键，然后使用 PORTB 驱动了一个七段数码管，表 5.28 为实例中使用的 Proteus 器件。

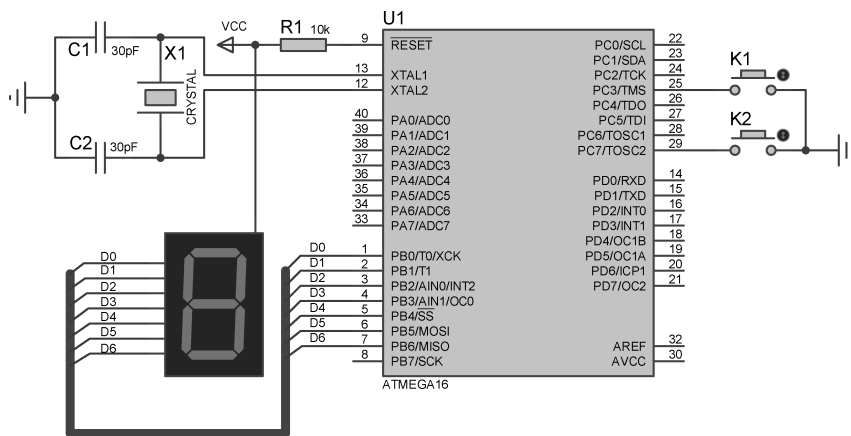


图 5.20 实例的 Proteus 电路

表 5.28 Proteus 电路器件列表

器件名称	库	子库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
BUTTON	Switches & Relays	Switches	独立按键
7SEG - COM - ANODE	Optoelectronics	7 - Segment Displays	七段共阳极数码管

### 5. 实例的应用代码

实例的应用代码如例 5.4 所示。

代码使用了一个 unsigned char 类型的软件计数器 Counter 作为数码管显示参数，为了避免在按键被按下时该计数器的值一直在被修改（即在每按一次键时计数器值只修改一次），使用了两个标志位 Aflg 和 Sflg 分别对应 K1 和 K2 被按下事件，然后在主循环中对该标志位进行相应的处理。

为了消除按键的抖动，代码还用了一个变量 KeyNum 来分两次检测按键的状态，其间间





隔一个非常微小的延时（通常几毫秒就足够了），如果两次按键的状态相同，则表明按键状态确实改变了，而不是一个意外的抖动。

#### 【例 5.4】 按键计数

```
#include <iom16v.h>
#include <macros.h>

#define TRUE 0xFF
#define FALSE 0x00
//字形编码
unsigned char const
SEGtable[ ] = { 0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x90, 0x88, 0x83, 0xc6, 0xa1,
0x86, 0x8e };

void delay( void)                //延时函数
{
    unsigned int i;
    for( i = 1; i < 100; i ++ );
}

void delay_1ms( void)            //1ms 延时函数
{
    unsigned int i;
    for( i = 1; i < ( unsigned int ) ( 8 * 143 - 2 ); i ++ );
}

void DelayMs( unsigned int time) //ms 延时函数
{
    unsigned int i = 0;
    while( i < time )
    {
        delay_1ms( );
        i ++ ;
    }
}

void port_init( void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0xFF;
    PORTC = 0x00;                //m103 output only
    DDRC  = 0x00;
```

```
PORTD = 0x00;
DDRD  = 0x00;
}
void init_devices( void)
{
    CLI( );
    port_init( );

    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI( );
}
void main( void)
{
    unsigned char KeyNum = 0x00;
    unsigned char temp;
    unsigned char Aflg = FALSE;
    unsigned char Sflg = FALSE;
    unsigned char Counter = 0;    //计数器
    init_devices( );
    while(1)
    {
        PORTC = 0xFF;    //上拉电阻
        KeyNum = PINC;    //读取 PC 的状态
        if( KeyNum != 0xFF)    //如果有按键被按下
        {
            DelayMs( 10 );    //延迟 10ms
            temp = PINC;    //再次读取 KeyNum
            if( KeyNum == temp)    //如果有按键被按下，读取按键状态
            {
                if( KeyNum == 0xF7)
                {
                    Aflg = TRUE;    //增加键被按下
                }
                if( KeyNum == 0x7F)
                {
                    Sflg = TRUE;    //减少键被按下
                }
            }
        }
        else
        {

```



```

        KeyNum = 0x00;          //有抖动延时, 被清除
    }
}
else                            //对按键进行处理
{
    if( Aflg == TRUE)
    {
        Aflg = FALSE;
        if( Counter == 9)        //如果已经加到顶了
        {
            Counter = 0;         //清除
        }
        else
        {
            Counter ++ ;
        }
    }
    if( Sflg == TRUE)
    {
        Sflg = FALSE;
        if( Counter == 0)        //如果已经减到最小了
        {
            Counter = 9;         //清除
        }
        else
        {
            Counter -- ;
        }
    }
}
PORTB = SEGtable[ Counter ];
}
}

```

## 6. 实例的仿真结果和说明

点击运行, 可以看到数码管显示相应的计数值, 分别按下 K1 和 K2, 可以看到数码管显示值的改变, 如图 5.21 所示。



### 总结

独立按键是键盘的基本组成部分, 除了按下和释放这两个状态之外, 还可以有长按、短按等状态, 读者可以自行研究如何构造这些状态的判别函数。

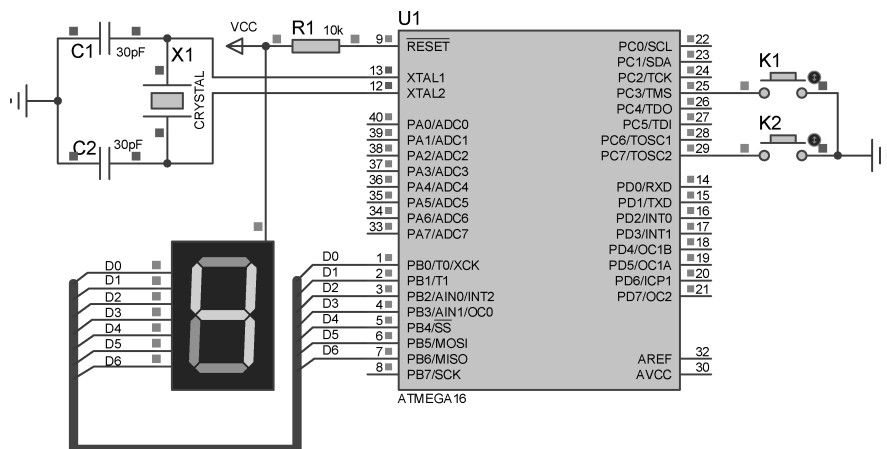


图 5.21 实例的仿真运行结果

### 5.4.5 I/O 引脚驱动行列键盘实例

和发光二极管类似，当在同一个 ATmega16 应用系统中需要使用多个独立按键时，固然可以使用多个 I/O 端口来驱动的实现方法，但这会有占用硬件资源过多的缺点，此时可以使用行列扫描键盘。本实例是一个使用 ATmega16 驱动行列扫描键盘，并且在数码管上显示对应的键盘值的应用。

#### 1. 行列键盘基础

行列扫描键盘是将很多的独立按键按照行、列的结构组合起来构成的一个整体键盘，这样可以减少对 ATmega16 的 I/O 引脚的占用数目，其组成结构如图 5.22 所示。

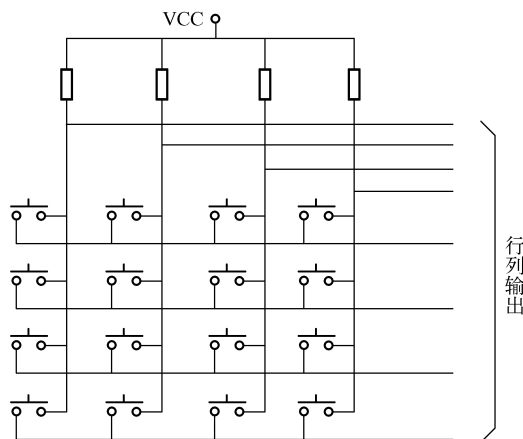


图 5.22 行列扫描键盘的组成结构

行列扫描键盘把独立的按键跨接在行扫描线和列扫描线之间，这样  $M \times N$  个按键就只需要  $M$  根行线和  $N$  根列线，大大减少了 I/O 引脚的占用，这样的行列扫描键盘被称为  $M \times N$  行列键盘。

## 2. Proteus 中的行列键盘

Proteus 中的行列扫描键盘位于 Switches & Relays 库的 Keypads 子分类库中, 提供了 KEYPAD - CALCULATOR、KEYPAD - PHONE 等多种行列扫描键盘, 如图 5.23 所示, 其中最常用的是 KEYPAD - SMALLCALC, 这是一个  $4 \times 4$  的 16 键行列扫描键盘。

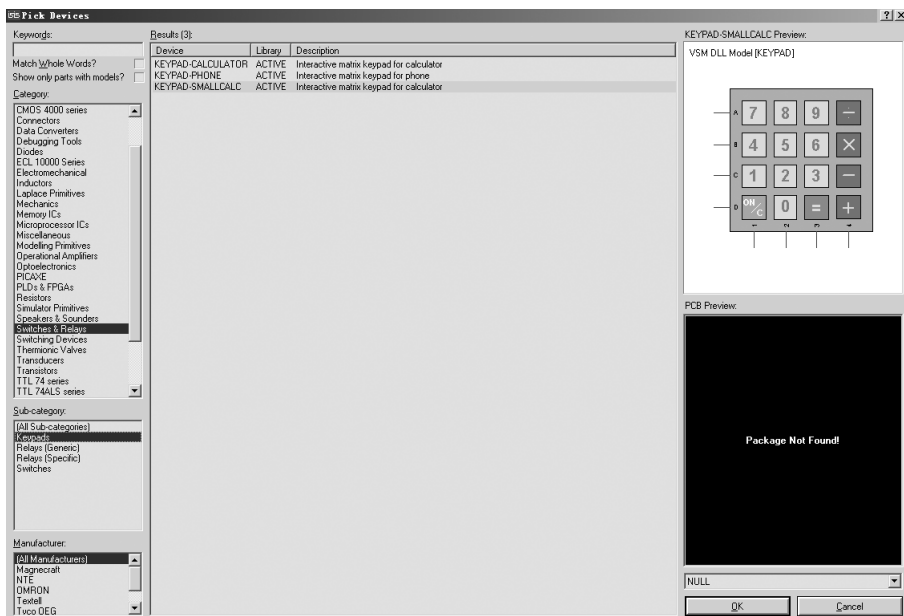


图 5.23 Proteus 中的行列扫描键盘

双击 KEYPAD - SMALLCALC, 弹出如图 5.24 所示的属性设置对话框, 在实际使用中不用关注。

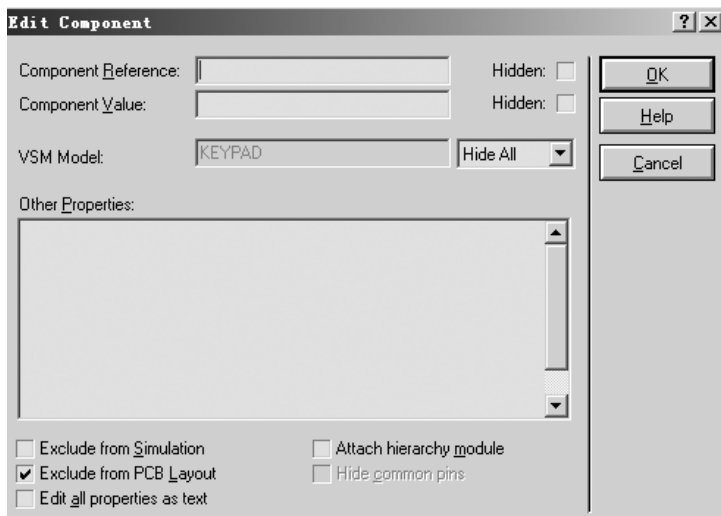


图 5.24 行列扫描键盘的属性设置对话框

### 3. 实例设计思路

在 ATmega16 应用系统中，通常使用行列扫描法来读取行列扫描键盘的按键状态，行列扫描法将行列扫描键盘的行线和列线分别连接到 ATmega16 的外部 I/O 引脚，然后进行如下操作。

- (1) 将所有的行线都置为高电平。
- (2) 依次将所有的列线都置为低电平，然后读取行线状态。
- (3) 如果对应的行列线上有按键被按下，则读入的行线为低电平。
- (4) 根据行列键盘的输出将按键编码并且输出。

当扫描到对应的按键之后，则对其进行相应的处理，可以参考应用实例 5.4 的设计来驱动数码管显示对应的数字或者字母。

### 4. 实例的 Proteus 电路图

实例的 Proteus 电路如图 5.25 所示，ATmega16 使用 PORTC 端口驱动了一个  $4 \times 4$  的行列扫描键盘，然后使用 PORTB 驱动了一个七段数码管，表 5.29 为实例中使用的 Proteus 器件。

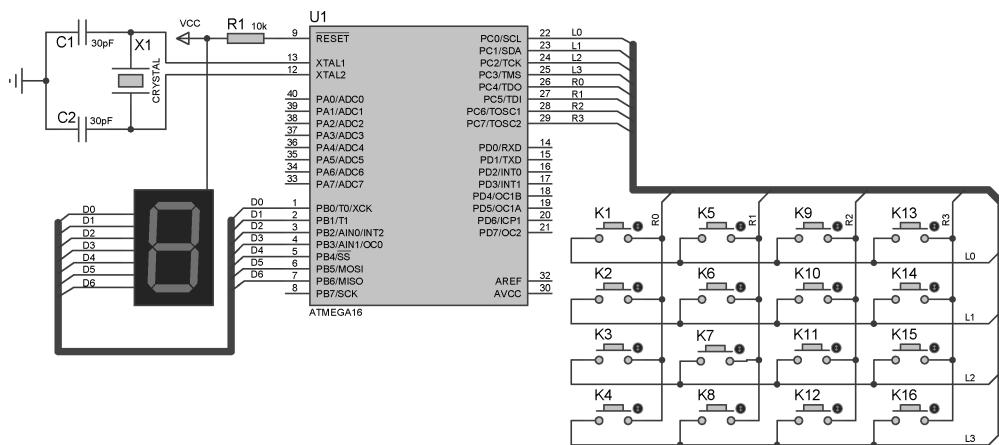


图 5.25 实例的 Proteus 电路

表 5.29 Proteus 电路器件列表

器件名称	库	子库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
BUTTON	Switches & Relays	Switches	独立按键
7SEG - COM - ANODE	Optoelectronics	7 - Segment Displays	七段共阳极数码管



## 5. 实例的应用代码

实例的应用代码如例 5.5 所示。

代码设计了一个 unsigned int 类型的函数 Key\_num，用于对键盘进行扫描并且返回按键编码，在调用 Key\_scan 扫描到按键值之后，可以通过查找对应的字形编码的方法把对应的字形代码送 PORTB 端口显示。

### 【例 5.5】 计算器键盘显示

```
#include <iom16v.h>
#include <macros.h>

unsigned int Key_num=0;
unsigned char const SEGtable[] =
{
    0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,
    0x88,0x83,0xc6,0xa1,0x86,0x8e,0xbf,0xff //共阳极 0. 1. 2. 7. 4. 5. 6. 7. 8. 9. A. b. C. d. E. F. -
};

void delay( void) //延时函数
{
    unsigned int i;
    for(i = 1;i < 100;i ++ );
}

void delay_1ms( void) //1ms 延时函数
{
    unsigned int i;
    for(i = 1;i < ( unsigned int ) ( 8 * 143 - 2 );i ++ );
}

void DelayMs(unsigned int time) //ms 延时函数
{
    unsigned int i=0;
    while( i < time)
    {
        delay_1ms();
        i ++ ;
    }
}

void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0x00;
```

```

PORTB = 0x00;
DDRB  = 0xFF;
PORTC = 0x00;
DDRC  = 0xf0;          //C 口低四位输入,置高电平,高四位输出,置低电平
PORTD = 0x00;
DDRD  = 0x00;
}

unsigned char Key_scan( void)          //键盘扫描函数
{
    unsigned char i,j;
    DDRC = 0xf0;          //设置 PD 高四位为输出口,低四位为输入口
    PORTC = 0x0f;          // 初始运行输出全为 0x0f
    if( ( PINC & 0x0f) == 0x0f) return 20;  //判断有无按键动作,没有,返回 20
    else
    {
        DelayMs(10);      //按键消抖
        if( ( PINC & 0x0f) == 0x0f) return 20; //再次判断是否有按键动作
        else
        {
            for(i=4;i < 8;i++)          //逐行输出 0
            {
                PORTC = ~(1 << i) | 0x0f;    //第 i 行输出 0
                for(j=0;j < 4;j++)
                {
                    if( ( PINC & (1 << j)) == 0)    //逐列检测
                    {
                        Key_num = (i-4) * 4 + j;    //计算键值
                    }
                }
            }
            return Key_num;          //返回键值
        }
    }
}

void Led_display( void)          //显示函数
{
    PORTB = SEGtable[ Key_num + 1 ];    //查表对应的显示编码,送出
}

void init_devices( void)
{
    CLI();
    port_init();
    MCUCR = 0x00;
}

```



```

GICR  =0x00;
TIMSK =0x00;
SEI();
}

void main(void)
{
    init_devices();
    PORTB =0xFF;
    while(1)
    {
        Key_scan();           //按键扫描
        Led_display();        //显示
    }
}

```

## 6. 实例的仿真结果和说明

点击运行，可以看到数码管显示相应的计数值，分别按下对应的按键，可以看到数码管输出显示值的改变，如图 5.26 所示。

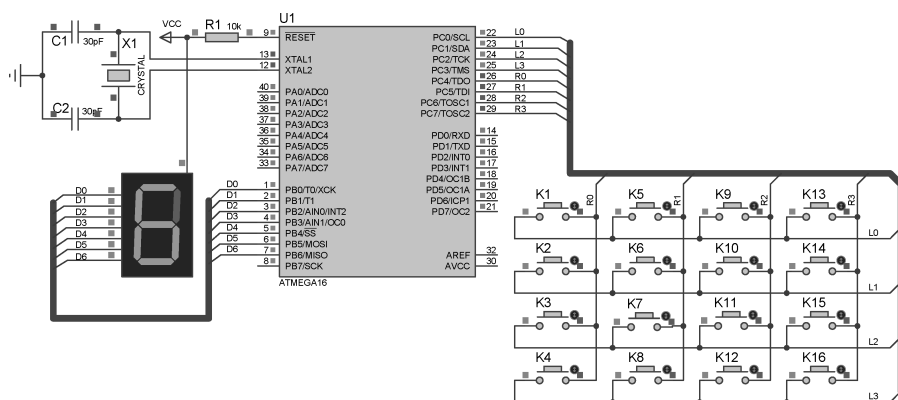


图 5.26 实例的仿真运行结果



## 总结

和独立按键类似，行列扫描键盘也有长按、短按等状态，此外还有组合按键状态（如同时按下“1”和“4”），读者可以自行研究如何构造这些状态的判别函数。

### 5.4.6 外部中断控制 I/O 引脚输出实例

本应用是一个使用 ATmega16 的外部中断来控制 I/O 引脚输出的实例，当检查到一个外部中断事件之后，ATmega16 将输出 I/O 引脚上的电平翻转。



### 1. 实例设计思路

设置好 ATmega16 的外部中断触发方式和 I/O 引脚为输出之后，等待外部中断事件，在外部中断服务子函数中将对应的引脚电平翻转即可。

### 2. 实例的 Protues 电路图

实例的 Proteus 电路如图 5.27 所示，在 INT0（PD2）引脚上连接了一个按键，当按键被按下时，对应的中断引脚被连接到低电平，在 PA7 引脚上连接了一个电压探针用于观测对应的电平状态，表 5.30 给出了实例中使用的 Proteus 器件。

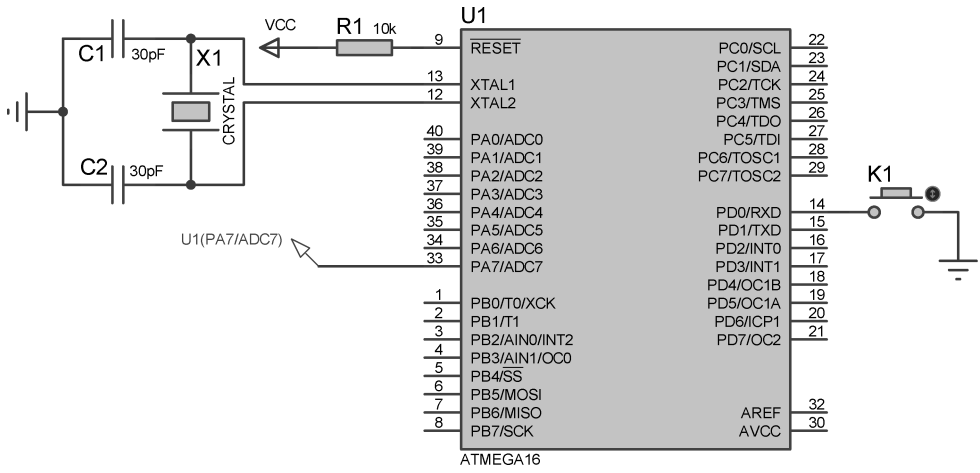


图 5.27 实例的 Proteus 电路

表 5.30 Proteus 电路器件列表

器件名称	大 类 库	子 类 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
BUTTON	Switches & Relays	Switches	独立按键

### 3. 实例的应用代码

应用实例的代码如例 5.6 所示，代码首先初始化 PORTA 端口，然后设置外部中断 0 为脉冲方式触发。

#### 【例 5.6】 外部中断控制 I/O 引脚输出

```
#include <iom16v.h>
#include <macros.h>
```



```
void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0x80;
    PORTB = 0x00;
    DDRB = 0x00;
    PORTC = 0x00;
    DDRC = 0x00;
    PORTD = 0x00;
    DDRD = 0x00;
}
//外部中断 0 的服务子程序
#pragma interrupt_handler int0_isr:iv_INT0
void int0_isr( void)
{
    PORTA ^= BIT(7);          //电平翻转
}

void init_devices( void)
{
    CLI();
    port_init();

    MCUCR = 0x02;
    GICR = 0x40;
    TIMSK = 0x00;
    SEI();
}

void main( void)
{
    init_devices();
    //insert your functional code here...
    while(1)
    {
    }
}
```

#### 4. 实例的仿真结果和说明

点击运行，按下按键，可以看到电压探针上的电平状态发生变化，如图 5.28 所示。

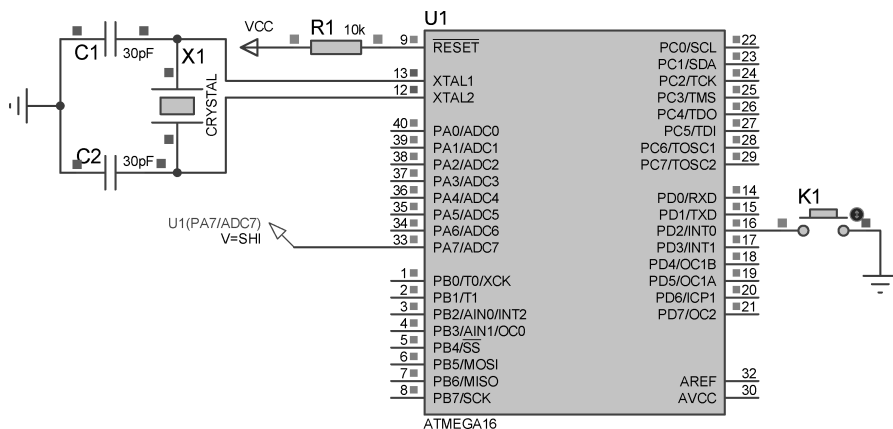


图 5.28 实例的仿真运行结果



## 总结

读者可以自行尝试修改 ATmega16 的中断触发方式。

## 第6章 ATmega16 单片机的定时计数器

ATmega16 提供了 T/C0、T/C1 和 T/C2 共三个定时/计数器，其功能非常强大，可以用于内部定时计数脉冲捕获等，本章详细介绍它们的使用方法。

### 6.1 定时计数器 T/C0

定时计数器 T/C0 是一个通用的单通道 8 位定时/计数器，可以作为频率发生器，也可以作为外部事件计数器，还可以作为 10 位的预分频器，它有溢出和比较匹配中断源（TOV0 和 OCF0），还可以提供无干扰脉冲、相位正确的 PWM 输出。

#### 6.1.1 T/C0 的相关寄存器

ATmega16 通过 T/C0 相关寄存器对其进行控制，这些寄存器包括 T/C0 控制寄存器（TCCR0）、T/C0 寄存器（TCNT0）、T/C0 输出比较寄存器（OCR0）、T/C0 中断屏蔽寄存器（TIMSK）、T/C0 中断标志寄存器（TIFR）以及特殊功能 I/O 寄存器（SFIOR）。

##### 1. T/C0 控制寄存器 TCCR0

TCCR0 寄存器用于对 T/C0 进行相应设置和控制，其内部结构如表 6.1 所示，每个位的功能说明如下。

表 6.1 T/C0 控制寄存器 TCCR0 内部结构

BIT	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00
读/写	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- FOC0：强制输出比较位，仅在当 WGM00 指明 T/C0 工作于非 PWM 模式时才有效。但是为了保证与未来器件的兼容，在使用 PWM 功能时 TCCR0 必须清零，对其置“1”后，波形发生器将立即进行比较操作，比较匹配输出引脚 OC0 将按照 COM01 ~ COM00 的设置输出相应的电平，需要注意的是，FOC0 类似一个锁存信号，真正对强制输出比较起作用的是 COM01 ~ COM00 的设置。FOC0 位不会引发任何中断，也不会在利用 OCR0 作为 TOP 的 CTC 模式下对定时器进行清零的操作，对 FOC0 的读操作的返回值永远为 0。
- WGM01 ~ WGM00：工作模式控制位，用于控制 T/C0 的计数序列、计数器的最大值 TOP，以及产生何种波形，其详细说明如表 6.2 所示。



表 6.2 工作模式控制位

编号	WGM01	WGM00	工作模式	TOP	OCR0 更新时间	TOV0 的置位时刻
0	0	0	普通	0xFF	立即更新	MAX
1	0	1	PWM, 相位修正	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	立即更新	MAX
3	1	1	快速 PWM	0xFF	TOP	MAX

- COM01 ~ COM00：比较匹配输出模式控制位，用于控制比较匹配发生时输出引脚 OC0 上的电平变化，如果 COM01 ~ COM00 中的一位或全部都被置位，OC0 以比较匹配输出的方式进行工作。同时其方向控制寄存器位要设置为“1”，以使能输出驱动器。当 OC0 被连接到物理引脚上时，COM01 ~ COM00 的功能依赖于 WGM01 ~ WGM00 的设置，表 6.3 至表 6.5 为当 WGM01 ~ WGM00 设置不同模式时 COM01 ~ COM00 的功能。

表 6.3 比较输出工作模式，非 PWM 模式

COM01	COM00	说 明
0	0	正常的端口操作，不和 OC0 连接
0	1	比较匹配发生时 OC0 取反
1	0	比较匹配发生时 OC0 清零
1	1	比较匹配发生时 OC0 置位

表 6.4 比较输出工作模式，快速 PWM 模式

COM01	COM00	说 明
0	0	正常的端口操作，不和 OC0 连接
0	1	保留
1	0	比较匹配发生时 OC0A 清零，计数到 TOP 时 OC0 置位
1	1	比较匹配发生时 OC0A 置位，计数到 TOP 时 OC0 清零



## 注意

当 OCR0 等于 TOP，且 COM01 被置位时，比较匹配将被忽略，而计数到 TOP 时 OC0 的动作将继续有效。

表 6.5 比较输出工作模式，相位修正 PWM 模式

COM01	COM00	说 明
0	0	正常的端口操作，不和 OC0 连接
0	1	保留
1	0	在升序计数时发生比较匹配将 OC0 清零，降序计数时置位 OC0
1	1	在升序计数时发生比较匹配将 OC0 置位，降序计数时清零 OC0

**注意**

当 OCR0 等于 TOP，且 COM01 被置位时，比较匹配将被忽略，而计数到 TOP 时 OC0 的动作将继续有效。

- CS02 ~ CS00：时钟选择位，用于选择 T/C0 的时钟源，其使用方法如表 6.6 所示。

表 6.6 T/C0 的时钟选择

CS02	CS01	CS00	说 明
0	0	0	无时钟，T/C0 不工作
0	0	1	CLK <sub>10</sub>
0	1	0	CLK <sub>10</sub> /8
0	1	1	CLK <sub>10</sub> /64
1	0	0	CLK <sub>10</sub> /256
1	0	1	CLK <sub>10</sub> /1024
1	1	0	时钟由 T0 引脚输入，下降沿触发
1	1	1	时钟由 T0 引脚输入，上升沿触发

**注意**

如果 T/C0 使用外部时钟，即使 T0 被配置为输出，其引脚上加的电平变化依然会驱动计数器，利用这个特性可以使用软件来计数。

**2. T/C0 寄存器 TCNT0**

TCNT0 寄存器用于对计数器的 8 位数据进行读写，对 TCNT0 寄存器的写访问将在下一个时钟阻止比较匹配。在计数器运行的过程中修改 TCNT0 的数值有可能丢失一次 TCNT0 和 OCR0 的比较匹配，TCNT0 寄存器的内部位结构如表 6.7 所示。

表 6.7 T/C0 寄存器 TCNT0 内部位结构

BIT	TCNT7	TCNT6	TCNT5	TCNT4	TCNT3	TCNT2	TCNT1	TCNT0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

**3. T/C0 输出比较寄存器 OCR0**

输出比较寄存器 OCR0 内包含一个 8 位的数据，不间断地与计数器数值 TCNT0 进行比较，匹配事件可以用来产生输出比较中断，或者用来在 OC0 引脚上产生波形，其内部位结构如表 6.8 所示。

表 6.8 T/C0 输出比较寄存器 OCR0 内部位结构

BIT	OCR7	OCR6	OCR5	OCR4	OCR3	OCR2	OCR1	OCR0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0



#### 4. T/C0 中断屏蔽寄存器 TIMSK

TIMSK 中的 OCIE0 位和 TOIE0 位用于对 T/C0 的中断使能事件进行处理，TIMSK 寄存器的内部结构如表 6.9 所示。

表 6.9 中断屏蔽寄存器 TIMSK 内部结构

BIT	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- OCIE0: T/C0 输出比较匹配中断使能位，当 OCIE0 和状态寄存器的全局中断使能位 I 都为“1”时，T/C0 的输出比较匹配中断使能，当 T/C0 的比较匹配发生，即 TIFR 中的 OCF0 置位时，触发中断事件。
- TOIE0: T/C0 溢出中断使能位，当 TOIE0 和状态寄存器的全局中断使能位 I 都为“1”时，T/C0 的溢出中断使能，如果有 T/C0 发生溢出，即 TIFR 中的 TOV0 位被置位时，触发中断事件。

#### 5. T/C0 中断标志寄存器 TIFR

T/C0 中断标志寄存器 TIFR 中的 OCF 位和 TOV0 位用于标志 T/C0 的中断事件，其内部结果如表 6.10 所示。

表 6.10 中断标志寄存器 TIFR 内部结构

BIT	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- OCF0: 输出比较标志 0 位，当 T/C0 与 OCR0（输出比较寄存器 0）的值匹配时，OCF0 被置位，此位在中断服务程序里被硬件清零，也可以通过对其写“1”来清零，当 SREG 中的位 I、OCIE0（T/C0 比较匹配中断使能）和 OCF0 位都被置位时，触发中断服务事件。
- TOV0: T/C0 溢出标志位，当 T/C0 溢出时，TOV0 被置位，此位在中断服务子程序中被硬件清零，TOV0 也可以通过写“1”来清零，当 SREG 中的位 I、TOIE0（T/C0 溢出中断使能位）和 TOV0 都被置位时，触发中断时间。当 ATmega16 工作在相位修正 PWM 模式下时，当 T/C0 在 0x00 改变计数方向时，TOV0 被置位。

#### 6. 特殊功能 I/O 寄存器 SFIOR

特殊功能 I/O 寄存器 SFIOR 的 PSR10 位控制了 T/C0 的预分频器复位操作，SFIOR 寄存器的内部结构如表 6.11 所示。

表 6.11 特殊功能 I/O 寄存器 SFIOR 内部结构

BIT	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10
读/写	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0



- PSR10: T/C1 与 T/C0 的预分频器复控制位, 置位时 T/C1 与 T/C0 的预分频器复位。操作完成后这一位由硬件自动清零。写入“0”时不会引发任何动作, T/C1 与 T/C0 共用同一个预分频器, 且预分频器复位对两个定时器均有影响, 对该位的读永远为“0”。

### 6.1.2 T/C0 的工作模式

T/C0 有 4 种工作模式: 普通工作模式、比较匹配时清零定时器 (CTC) 工作模式、快速 PWM 工作模式和相位修正 PWM 工作模式。T/C0 的工作模式由 WGM01、WGM00 和 COM01、COM00 位来控制, 其中比较输出工作模式对计数序列没有影响, 而波形产生工作模式则对计数序列有影响。COM01 和 COM00 位控制 PWM 的输出是否为反极性, 在非 PWM 工作模式下 COM01 ~ COM00 则控制输出是否应该在比较匹配发生时置位、清零或是电平取反。

#### 1. 普通工作模式

T/C0 的普通工作模式 (WGM01 ~ WGM00 = 00) 是 T/C0 的最简单的工作模式, 在此模式下计数器不停地累加, 当累积到寄存器的最大值后 (TOP = 0xFF), 由于数值溢出, 计数器简单地返回到最小值 0x00 重新开始计数。当 TCNT0 寄存器为零时的同一个定时器时钟里 T/C0 溢出标志 TOV0 被置位, 此时 TOV0 类似 TCNT0 的第 9 位, 只是只能置位, 不能清零。但由于定时器中断服务程序能够自动清零 TOV0, 因此可以通过软件提高定时器的分辨率。



#### 注意

在普通工作模式下, 可以向 TCNT0 寄存器随时写入新的计数器数值, 而输出比较单元可以用来产生中断, 但是不推荐在普通模式下利用输出比较来产生波形, 因为这会占用太多的 ATmega16 操作时间。

#### 2. 比较匹配时清零定时器 (CTC) 工作模式

在 CTC 工作模式 (WGM01 ~ WGM00 = 01) 下, OCR0 寄存器用于调节计数器的分辨率。当计数器 TCNT0 的数值等于 OCR0 寄存器的数值时, 计数器被清零, OCR0 定义了计数器的 TOP 值, 即计数器的分辨率。

该工作模式使得用户可以很容易地控制比较匹配输出的频率, 也简化了外部事件计数的操作。CTC 工作模式的时序图为如图 6.1 所示, 计数器值 TCNT0 一直累加到其数值与 OCR0 的数值匹配为止, 然后 TCNT0 被清零。

利用 OCF0 标志位可以在计数器数值达到 TOP 时产生中断, 在中断服务子程序里可以更新 TOP 的数值, 但是由于在 CTC 工作模式下 OCR0 寄存器没有双缓冲功能, 所以在计数器工作在无预分频器或很低的预分频器的状态下时, 将 TOP 更改为接近 BOTTOM 的数值时要注意。如果写入的 OCR0 寄存器的数值小于当前 TCNT0 寄存器值, 计数器将丢失一次比较匹配。在下一次比较匹配发生之前, 计数器不得不先计数到最大值 0xFF, 然后再从 0x00 开始计数到 OCR 的值。

为了在 CTC 工作模式下得到波形输出, 可以设置 OC0 在每次比较匹配发生时改变逻辑电平。这可以通过设置 COM01 和 COM00 位为“01”来完成, 在期望获得 OC0 的输出之前,

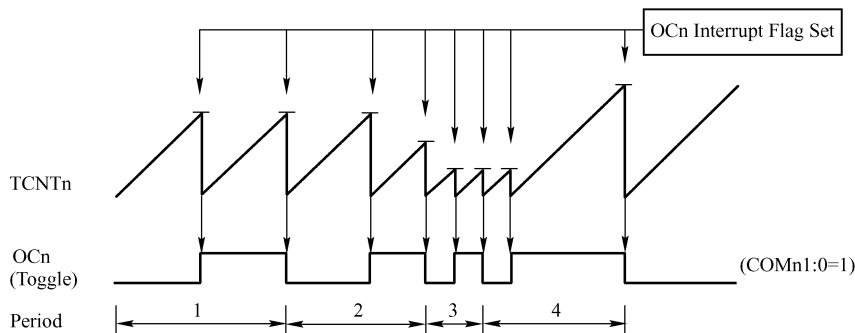


图 6.1 CTC 工作模式的时序图

首先要将其端口设置为输出。波形发生器能够产生的最大频率为：

$$f_{oc0} = \frac{f_{clk10}}{2} (\text{OCR0} = 0x00)$$

其频率由如下公式确定：

$$f_{oc0} = \frac{f_{clk10}}{2 \times N \times (1 + \text{OCRn})}$$

其中，变量  $N$  代表预分频因子，可以为 1、8、64、256 或者 1024，在普通模式下，TOV0 标志的置位发生在计数器从 MAX 变为 0x00 的时钟周期。

### 3. 快速 PWM 工作模式

T/C0 的快速 PWM 工作模式（WGM01 ~ WGM00 = 11）可用来产生高频的 PWM 波形。快速 PWM 模式与其他 PWM 模式的不同之处是其具有单斜坡工作方式：计数器从 BOTTOM 计到 MAX，然后立即回到 BOTTOM 重新开始。对于普通的比较输出模式，输出比较引脚 OC0 在 TCNT0 与 OCR0 匹配时清零，在 BOTTOM 时置位；对于反向比较输出模式，OC0 的动作正好相反。由于使用了单斜坡模式，快速 PWM 工作模式的工作频率比使用双斜坡的相位修正 PWM 工作模式高一倍，此高频操作特性使得快速 PWM 工作模式十分适合于功率调节、整流和 DAC 应用。较高的频率可以减小外部元器件（电感、电容）的物理尺寸，从而可以降低系统成本。

当 T/C0 处于快速 PWM 工作模式时，计数器的数值一直增加到 MAX，然后在后面的一个时钟周期清零。具体的时序图如图 6.2 所示，图中柱状的 TCNT0 表示这是单边斜坡操作，方框图时则包含了普通的 PWM 输出以及反向 PWM 输出，TCNT0 斜坡上的短水平线表示 OCR0 和 TCNT0 的比较匹配。

当计时器数值达到 MAX 时，T/C0 的溢出标志 TOV0 被置“1”，此时如果中断被使能，则在中断服务程序可以更新比较值。

当 T/C0 处于快速 PWM 工作模式时，比较单元可以在 OC0 引脚上输出 PWM 波形，设置 COM01 ~ COM00 为“10”则可以产生普通的 PWM 信号，为“11”则可以产生反向 PWM 波形。

如果要想在引脚上得到输出信号，还必须将 OC0 引脚的数据方向设置为输出，产生

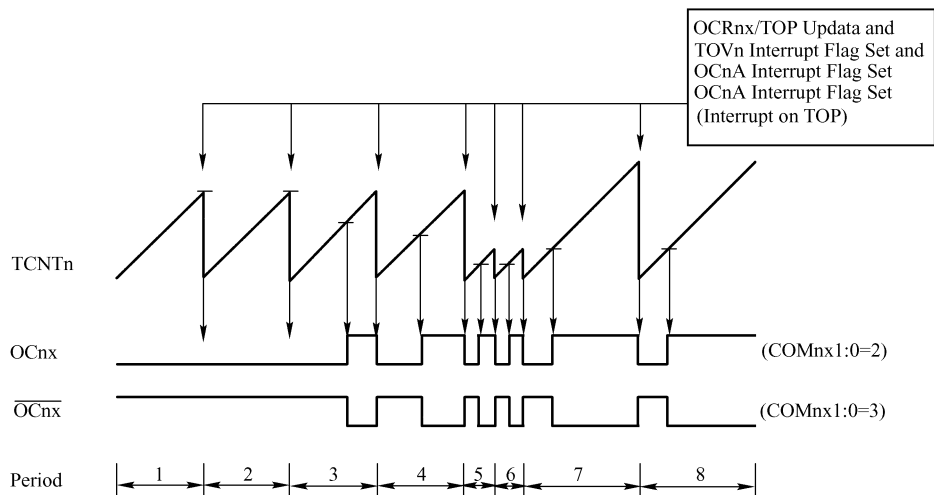


图 6.2 快速 PWM 工作模式的时序图

PWM 波形的原理是，OC0 寄存器在 OCR0 与 TCNT0 匹配时被置位（或被清零），以及在计数器清零（从 MAX 变为 BOTTOM）的那一个定时器时钟周期被清零（或被置位）。

在快速 PWM 工作模式下，输出的 PWM 频率可以通过如下公式计算得到：

$$f_{\text{OCnPWM}} = \frac{f_{\text{clkIO}}}{N \times 256}$$

其中，变量  $N$  代表分频因子 1、8、64、256 或者 1024。

当 OCR0 寄存器为极限值时，表示了快速 PWM 工作模式的一些特殊情况，若 OCR0 等于 BOTTOM，输出则为出现在第 MAX + 1 个定时器时钟周期的窄脉冲，若 OCR0 等于 MAX 时，则根据 COM01 ~ COM00 的设定，输出恒为高电平或低电平。

通过设定 OC0 引脚在比较匹配时进行逻辑电平取反（COM01 ~ COM00 = 1），可以得到占空比为 50% 的周期信号，当 OCR0 寄存器为“0”时，该信号有最高频率：

$$f_{\text{oc0}} = \frac{f_{\text{clkIO}}}{2} (\text{OCR0} = 0x00)$$

这个特性类似于 CTC 工作模式下的 OC0 取反操作，不同之处是快速 PWM 模式具有双缓冲功能。

#### 4. 相位修正 PWM 工作模式

相位修正 PWM 工作模式（WGM01 ~ WGM00 = 1）为用户提供了一个获得高精度相位修正 PWM 波形的办法。此模式基于双斜坡操作，计时器重复地从 BOTTOM 计到 MAX，然后又从 MAX 倒退回到 BOTTOM。在一般的比较输出模式下，当计时器往 MAX 计数时，若 TCNT0 与 OCR0 匹配，OC0 将清零为低电平；而在计时器往 BOTTOM 计数时，若 TCNT0 与 OCR0 匹配，OC0 将置位为高电平。工作于反向输出比较模式时则正好相反，与单斜坡操作相比，双斜坡操作可获得的最大频率要小，但由于其对称的特性，十分适合于电机之类的控制。

相位修正 PWM 工作模式的 PWM 精度固定为 8bit，计时器不断地累加直到 MAX，然后开始减计数，在一个定时器时钟周期里 TCNT0 的值等于 MAX，其时序图如图 6.3 所示。图中 TCNT0 的数值用柱状图表示，以说明双斜坡操作，该图同时说明了普通 PWM 的输出和反向 PWM 的输出，TCNT0 斜坡上的短水平线则表示 OCR0 与 TCNT0 的比较匹配。

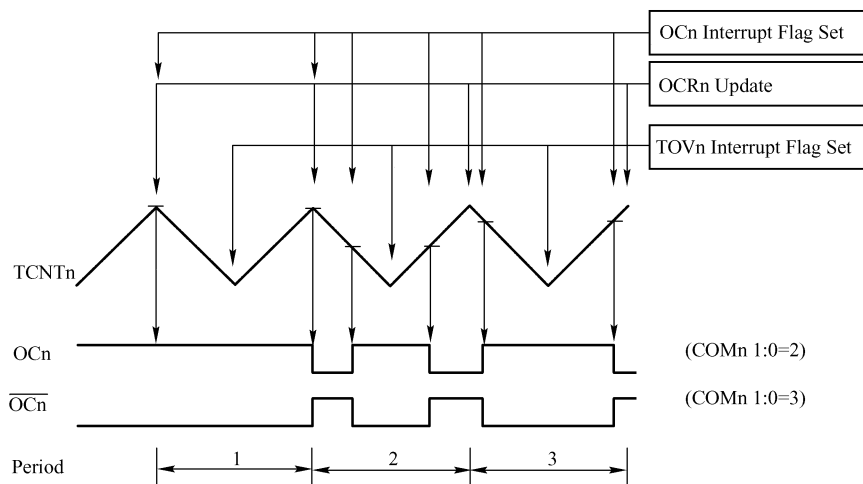


图 6.3 相位修正 PWM 工作模式的时序图

当计时器达到 BOTTOM 时，T/C0 的溢出标志位 TOV0 将被置位，此标志位可用来产生中断事件。

将 ATmega16 工作于相位修正 PWM 模式时，比较单元可以在 OC0 引脚产生 PWM 波形，将 COM01 ~ COM00 设置为“10”将产生普通相位的 PWM，设置 COM01 ~ COM00 为“11”则会产生反向 PWM 信号，如果要想在引脚上得到输出信号，必须将 OC0 引脚的数据方向设置为输出。当 OCR0 和 TCNT0 比较匹配发生时，OC0 寄存器将产生相应的清零或置位操作，从而产生 PWM 波形。

在 ATmega16 工作于相位修正 PWM 模式时，其 PWM 频率可由下式公式获得：

$$f_{\text{OCnPWM}} = \frac{f_{\text{clkIO}}}{N \times 510}$$

当 OCR0 寄存器处于极值时，代表了相位修正 PWM 模式的一些特殊情况，在普通 PWM 工作模式下，若 OCR0 等于 BOTTOM，则输出一直保持为低电平；若 OCR0 等于 MAX，则输出保持为高电平，反向 PWM 模式正好相反。

如图 6.3 所示，在第 2 个周期，虽然没有发生比较匹配，OCn 引脚上也出现了一个从高到低的跳变，其目的是保证波形在 BOTTOM 两侧的对称，没有比较匹配时有如下两种情况会出现跳变。

(1) OCR0A 的值从 MAX 改变为其他数据，当 OCR0A 值为 MAX 时，引脚 OCn 的输出应该与前面降序计数比较匹配的结果相同。为保证波形在 BOTTOM 两侧对称，当 T/C0 的数值为 MAX 时，引脚 OCn 的输出又必须符合后面升序计数比较匹配的结果。

(2) 定时器从一个比 OCR0A 高的值开始计数，并因此丢失了一次比较匹配，系统由此引入发生 OCn 却仍然有跳变的现象。



## 6.2 定时计数器 T/C1

ATmega16 的定时计数器 T/C1 是 16 位的, 通常用来实现需要精确的程序定时、波形产生以及信号测量, 其内置带双缓冲的输出比较寄存器、两个独立的输出比较单元、输入捕捉噪声抑制器、内部频率发生器、一个外部事件计数器和一个输入捕捉单元, 允许 16 位周期可变的 PWM, 并且支持 4 个独立的中断源 (TOV1、OCF1A、OCF1B 与 ICF1)。

### 6.2.1 T/C1 的相关寄存器

T/C1 的相关控制寄存器比较多, 包括 T/C1 控制寄存器 TCCR1A 和 TCCR1B, 数据寄存器 TCNT1H 和 TCNT1L, 输出比较寄存器 OCR1AH 和 OCR1AL, 输出比较寄存器 OCR1BH 和 OCR1BL, 输入捕捉寄存器 ICR1H 和 ICR1L, 中断屏蔽寄存器 TIMSK, 中断标志寄存器 TIFR。

#### 1. T/C1 控制寄存器 A——TCCR1A

TCCR1A 用于对 T/C1 的控制操作, 其内部结构如表 6.12 所示。

表 6.12 TCCR1A 寄存器内部结构

BIT	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
读/写	R/W	R/W	R/W	R/W	W	W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- COM1A1 ~ COM1A0 与 COM1B1 ~ COM1B0: OC1A 与 OC1B 状态控制位。如果一位或两位被写入“1”, OC1A (OC1B) 输出功能将取代 IO 端口功能。此时 OC1A (OC1B) 对应的外部引脚数据方向控制必须置位以使能输出驱动器。当 OC1A (OC1B) 与物理引脚相连时, 这些位的功能由 WGM13 ~ WGM10 的设置决定, 表 6.13 至表 6.15 是 WGM13 ~ WGM10 在不同的工作模式下对应的功能定义 (WGM13 ~ WGM12 在 TCCR1B 寄存器中)。

表 6.13 比较输入工作模式

COM1A1/COM1B1	COM1A0/COM1B0	说明
0	0	断开, 普通端口
0	1	比较匹配时 OC1A/OC1B 取反
1	0	比较匹配时清零 OC1A/OC1B
1	1	比较匹配时置位 OC1A/OC1B

表 6.14 快速 PWM 工作模式

COM1A1/COM1B1	COM1A0/COM1B0	说 明
0	0	断开, 普通端口
0	1	WGM13 ~ WGM10 = 15, 比较匹配时 OC1A 取反, OC1B 断开; WGM13 ~ WGM10 = 其他值, OC1A/OC1B 断开
1	0	比较匹配时清零 OC1A/OC1B, OC1A/OC1B 在 TOP 时置位
1	1	比较匹配时置位 OC1A/OC1B, OC1A/OC1B 在 TOP 时清零

表 6.15 相位修正以及相频修正 PWM 工作模式

COM1A1/COM1B1	COM1A0/COM1B0	说 明
0	0	断开，普通端口
0	1	WGM13~WGM10 = 9 或者 14，比较匹配时 OC1A 取反，OC1B 断开； WGM13~WGM10 = 其他值，OC1A/OC1B 断开
1	0	升序计数时比较匹配将清零 OC1A/OC1B，降序计数比较匹配将置位 OC1A/OC1B
1	1	升序计数时比较匹配将置位 OC1A/OC1B，降序计数比较匹配将清零 OC1A/OC1B

- FOC1A 和 FOC1B：通道 A/B 强制输出比较控制位。当 WGM13 ~ WGM10 指定为非 PWM 模式时这两位被激活，当 T/C1 工作在 PWM 模式下对 TCCR1A 写入时，这两位必须被置“0”。当 FOC1A 和 FOC1B 位被置“1”，立即强制波形产生单元进行比较匹配。COM1x1 ~ COM1x0 的设置改变 OC1A/OC1B 的输出。FOC1A/FOC1B 位作为选通信号。COM1x1：0 位的值决定强制比较的效果。在 CTC 工作模式下使用 OCR1A 作为 TOP 值，FOC1A/FOC1B 选通既不会产生中断，也不会清除定时器。对 FOC1A/FOC1B 位的读操作总是为 0。
- WGM11 ~ WGM10 位：波形发生模式控制位。这两位与位于 TCCR1B 寄存器的 WGM13 和 WGM12 位相结合，用于控制计数器的计数序列，如表 6.16 所示。

表 6.16 波形发生模式

模式	WGM13	WGM12	WGM11	WGM10	工作模式	TOP	OCR1x 更新时刻	TOV1 置位时刻
0	0	0	0	0	普通模式	0xFFFF	立即	MAX
1	0	0	0	1	8 位相位修正 PWM	0x00FF	TOP	BOTTOM
2	0	0	1	0	9 位相位修正 PWM	0x01FF	TOP	BOTTOM
3	0	0	1	1	10 位相位修整 PWM	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	立即	MAX
5	0	1	0	1	8 位快速 PWM	0x00FF	TOP	TOP
6	0	1	1	0	9 位快速 PWM	0x01FF	TOP	TOP
7	0	1	1	1	10 位快速 PWM	0x03FF	TOP	TOP
8	1	0	0	0	相位与频率修正 PWM	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	相位与频率修正 PWM	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	相位修正 PWM	ICR1	TOP	BOTTOM
11	1	0	1	1	相位修正 PWM	—	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	立即	MAX
13	1	1	0	1	保留	OCR1A	—	—
14	1	1	1	0	快速 PWM		TOP	TOP
15	1	1	1	1	快速 PWM		TOP	TOP



## 2. T/C 控制寄存器 B——TCCR1B

TCCR1B 也用于对 T/C1 的控制操作，其内部结构如表 6.17 所示。

表 6.17 TCCR1B 寄存器内部结构

BIT	ICNC1	ICES1	—	WGM13	WGM12	CS12	CS11	CS10
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- ICNC1：输入捕捉噪声抑制器控制位。置位 ICNC1 将使能输入捕捉噪声抑制功能，此时外部引脚 ICP1 的输入被滤波，其作用是从 ICP1 引脚连续进行 4 次采样，如果 4 个采样值都相等，那么信号送入边沿检测器，否则被抛弃；因此使能该功能使得输入捕捉被延迟了 4 个时钟周期。
- ICES1：输入捕捉触发沿选择位。该位控制选择使用 ICP1 上的哪个边沿触发捕获事件。ICES 位为“0”选择的是下降沿触发输入捕捉；ICES1 位为“1”选择的是逻辑电平的上升沿触发输入捕捉。当 ICES1 捕获到一个事件后，计数器的数值被复制到 ICR1 寄存器，该捕获事件还会置位 ICF1，如果中断被使能，输入捕捉事件即被触发。当 ICR1 被用作 TOP 值时，ICP1 与输入捕捉功能脱开，而输入捕捉功能被禁用。
- WGM13 ~ WGM12：波形发生模式控制位置，参考 TCCR1A 寄存器中的描述。
- CS12 ~ CS10：时钟源选择位。用于选择 T/C1 的时钟源。选择使用外部时钟源后，即使 T1 外部引脚被定义为输出，引脚上的逻辑信号电平变化仍然会驱动 T/C1 计数，这个特性允许用户通过软件来控制计数，如表 6.18 所示。

表 6.18 C/T1 的时钟选择

CS12	CS11	CS10	来 源
0	0	0	关闭 T/C1
0	0	1	CLK <sub>10</sub>
0	1	0	CLK <sub>10</sub> /8
0	1	1	CLK <sub>10</sub> /64
1	0	0	CLK <sub>10</sub> /256
1	0	1	CLK <sub>10</sub> /1024
1	1	0	T1 外部引脚，下降沿驱动
1	1	1	T1 外部引脚，上升沿驱动

## 3. T/C1 内容寄存器 TCNT1H 和 TCNT1L

TCNT1H 与 TCNT1L 组成了 T/C1 的数据寄存器 TCNT1，可以直接通过对它们读/写访问实现对 T/C1 这个 16 位计数器进行读/写访问，为了保证对高字节与低字节的同时读/写，必须使用一个 8 位的临时高字节寄存器 TEMP 来暂时保存数据，TCNT1 的内部结构如表 6.19 所示。





表 6.19 TCNT1H 和 TCNT1L 寄存器内部位结构

BIT	TCNT1H							
	TCNT1L							
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

**注意**

如果在 T/C1 运行期间修改 TCNT1 的内容，有可能会丢失一次 TCNT1 与 OCR1<sub>x</sub> 的比较匹配操作，因为写 TCNT1 寄存器将在下一个定时器周期阻塞比较匹配。

**4. T/C1 输出比较寄存器 1A 和 1B——OCR1AH、OCR1AL、OCR1BH、OCR1BL**

这两个寄存器中的 16 位数据与 TCNT1 寄存器中的计数值进行连续比较，一旦数据匹配，将产生一个输出比较中断，或改变 OC1<sub>x</sub> 的输出逻辑电平，输出比较寄存器长度均为 16 位，为保证对高字节与低字节的同时读写，也使用了一个 8 位临时高字节寄存器 TEMP，这两个寄存器的内部位结构如表 6.20 所示。

表 6.20 输出比较寄存器内部位结构

BIT	OCR1A							
	OCR1B							
	OCR1B							
	OCR1B							
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

**5. T/C1 输入捕捉寄存器 1——ICR1H 和 ICR1L**

当外部 I/O 引脚 ICP1（ATmega16 的模拟比较器模块的输出）有输入捕捉触发信号产生时，计数器 TCNT1 中的值被写入 ICR1 中，ICR1 的设定值可作为计数器的 TOP 值，ICR1 同样使用了 TEMP 来保证高低字节的同时读/写，其内部位结构如表 6.21 所示。

表 6.21 ICR1H 和 ICR1L 寄存器内部位结构

BIT	ICR1H							
	ICR1L							
读/写	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

**6. T/C1 的中断屏蔽寄存器 TIMSK**

T/C1 的中断屏蔽寄存器 TIMSK 用于对 T/C1 的相关中断进行控制，其内部位结构如表 6.22 所示，其中 TICIE1、OCIE1A、OCIEB、TOIE1 参与了对 T/C1 相关中断的控制。





表 6.22 TIMSK 中断屏蔽寄存器内部位结构

BIT	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- **TICIE1**: T/C1 输入捕捉中断使能位。当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，T/C1 的输入捕捉中断使能，如果 TIFR 寄存器的 ICF1 位被置位，即触发 T/C1 输入捕捉中断事件。
- **OCIE1A**: T/C1 输出比较 A 匹配中断使能位。当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，T/C1 的输出比较 A 匹配中断使能，如果 TIFR 寄存器的 OCF1A 位被置位，即触发 T/C1 输出比较 A 匹配中断事件。
- **OCIE1B**: T/C1 输出比较 B 匹配中断使能位。当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，T/C1 的输出比较 B 匹配中断使能，如果 TIFR 寄存器的 OCF1B 位被置位，即触发 T/C1 输出比较 B 匹配中断服务。
- **TOIE1**: T/C1 溢出中断使能位。当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，T/C1 的溢出中断使能，一旦 TIFR 寄存器的 TOV1 被置位，即触发 T/C1 溢出中断事件。

#### 7. T/C1 中断标志寄存器 TIFR

T/C1 的中断标志寄存器 TIFR 用于标志 T/C1 的相应中断触发事件，其内部位结构如表 6.23 所示，其中 ICF1、OCF1A、OCF1B 和 TOV1 位参与了 T/C1 的相应中断标志。

表 6.23 TIFR 中断标志寄存器内部位结构

BIT	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- **ICF1**: T/C1 输入捕捉标志位。当外部引脚 ICP1 出现捕捉事件时，ICF1 被置位。另外当 ICR1 作为计数器的 TOP 值，计数器值达到 TOP 时，ICF1 也被置位。当进入输入捕捉中断服务子程序时，ICF1 被硬件清零，也可以对其写入逻辑“1”来清除该标志位。
- **OCF1A**: T/C1 输出比较 A 匹配标志位。当 TCNT1 与 OCR1A 匹配成功时，该位被置位，强制输出比较（FOC1A）不会置位 OCF1A。当进入强制输出比较匹配 A 中断服务程序时，OCF1A 位被硬件自动清零，也可以对其写入逻辑“1”来清除该标志位。
- **OCF1B**: T/C1 输出比较 B 匹配标志位。当 TCNT1 与 OCR1B 匹配成功时，该位被置位“1”，强制输出比较（FOC1B）不会置位 OCF1B。当进入输出比较匹配 B 中断服务程序时，OCF1B 被硬件清零，也可以对其写入逻辑“1”来清除该标志位。
- **TOV1**: T/C1 溢出标志位。该位的设置与 T/C1 的工作方式有关，在普通工作模式和 CTC 工作模式下，当 T/C1 溢出时 TOV1 置位，在其他工作模式下的 TOV1 标志位置位，参考表 6.16，当进入中断服务子程序之后，该标志位被硬件清除。

## 6.2.2 T/C1 的工作模式

T/C1 共有 5 种工作模式：普通工作模式、比较匹配清零定时器（CTC）工作模式、快速 PWM 工作模式、相位修正 PWM 工作模式，相位与频率修正 PWM 工作模式。

### 1. 普通工作模式

在普通工作模式下，T/C1 计数器不停地累加，当计数到最大值后（TOP = 0xFFFF），由于数值溢出，计数器简单地返回到最小值 0x0000 重新开始。在 TCNT1 为零的同一个定时器时里，T/C1 溢出标志 TOV1 被置位，此时 TOV1 类似第 17 位，只是只能置位，不会清零。但由于 T/C1 的中断服务程序能够自动清零 TOV1 位，因此可以通过软件提高定时器的分辨率。在普通工作模式下没有什么需要特殊考虑的，ATmega16 可以随时写入新的计数器数值。

在普通工作模式下，输入捕捉单元很容易使用，要注意的是，外部事件的最大时间间隔不能超过计数器的分辨率，如果时间间隔太长，必须使用定时器溢出中断或预分频器来扩展输入捕捉单元的分辨率。

输出比较单元可以用来产生中断。但是不推荐在普通模式下利用输出比较来产生波形，因为这会占用太多的 ATmega16 处理器时间。

### 2. 比较匹配时清零定时器（CTC）工作模式

在 CTC 工作模式下，OCR1A 或 ICR1 寄存器用于调节计数器的分辨率，当计数器的数值 TCNT1 等于 OCR1A 或 ICR1 的值时，计数器清零。其中，OCR1A 或 ICR1 定义了计数器的 TOP 值，即计数器的分辨率。

在这个工作模式下，ATmega16 可以很容易地控制比较匹配输出的频率，也简化了外部事件计数的操作，CTC 工作模式的时序图如图 6.4 所示，计数器数值 TCNT1 一直累加到 TCNT1 与 OCR1 或 ICR1 匹配，然后 TCNT1 被清零。

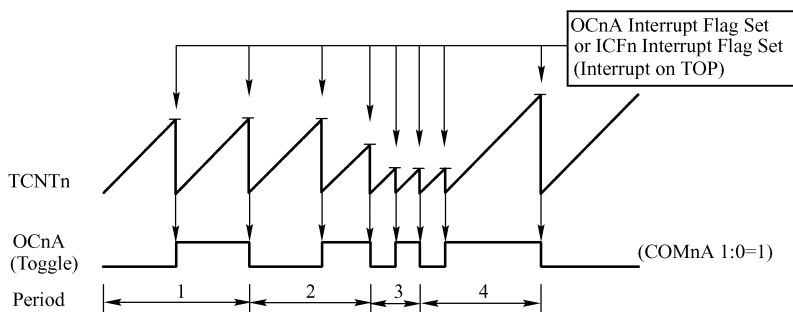


图 6.4 CTC 工作模式的时序图

利用 OCF1A 或 ICF1 标志可以在计数器数值达到 TOP 时产生中断，在中断服务程序里可以更新 TOP 的数值。由于 CTC 工作模式没有双缓冲功能，在计数器以无预分频器或很低的预分频器工作时，将 TOP 更改为接近 BOTTOM 的数值时要注意。如果写入的 OCR1A 或 ICR1 的数值小于当前 TCNT1 的数值，计数器将丢失一次比较匹配。在下一次比较匹配发生之前，计数器不得不先计数到最大值 0xFFFF，然后再从 0x0000 开始计数到 OCR1A 或 ICR1。在许多情况下，这一特性并非我们所希望的，替代的方法是使用快速 PWM 工作模式，该工



作模式使用 OCR1A 定义 TOP 值，因为此时 OCR1A 为双缓冲方式。

为了在 CTC 工作模式下得到波形输出，可以设置 OC1A 在每次比较匹配发生时改变逻辑电平，这可以通过设置 COM1A1 ~ COM1A0 = 1 来完成。在期望获得 OC1A 输出之前，首先要将其端口设置为输出（DDR\_OC1A = 1）。波形发生器能够产生的最大频率为  $f_{OC1A} = f_{clkIO}/2$ （OCR1A = 0x0000）。频率由如下公式确定：

$$f_{OCnA} = \frac{f_{clkIO}}{2 \times N \times (1 + OCRnA)}$$

其中，变量  $N$  代表预分频因子（1、8、64、256 或 1024）。

### 3. 快速 PWM 工作模式

快速 PWM 工作模式可用来产生高频的 PWM 波形，快速 PWM 工作模式与其他 PWM 工作模式的不同之处是其单边斜坡工作方式：计数器从 BOTTOM 计数到 TOP，然后立即回到 BOTTOM 重新开始。对于普通的比较输出模式，输出比较引脚 OC1x 在 TCNT1 与 OCR1x 匹配时置位，在 TOP 时清零；对于反向比较输出模式，OCR1x 的动作正好相反。由于使用了单边斜坡模式，快速 PWM 工作模式的工作频率比使用双斜坡的相位修正 PWM 工作模式高一倍。此高频操作特性使得快速 PWM 工作模式十分适合于功率调节、整流或者 DAC 应用。高频可以减小外部元器件（电感、电容）的物理尺寸，从而降低系统成本。

当 T/C1 处于快速 PWM 工作模式时，PWM 的分辨率可固定为 8、9 或 10 位，也可由 ICR1 或 OCR1A 来定义，其最小分辨率为 2 bit（ICR1 或 OCR1A 设为 0x0003），最大分辨率为 16 位（ICR1 或 OCR1A 设为 MAX），PWM 的分辨率位数可用下式计算：

$$R_{PWM} = \frac{\log(TOP + 1)}{\log(2)}$$

当 T/C1 处于快速 PWM 工作模式时，计数器的数值一直累加到固定数值 0x00FF、0x01FF、0x03FF、ICR1 或 OCR1A，然后在随后的一个时钟周期清零，其时序图如 6.5 所示。图中给出了当使用 OCR1A 或 ICR1 来定义 TOP 值时的快速 PWM 工作模式，图中柱状的 TCNT1 表示单边斜坡操作，方框图同时包含了普通的 PWM 输出以及反向 PWM 输出，TCNT1 斜坡上的短水平线表示 OCR1x 和 TCNT1 的匹配比较，在比较匹配后 OC1x 中断标志置位。

当计时器数值达到 TOP 时，T/C1 溢出标志 TOV1 被置位，另外若 TOP 值是由 OCR1A 或 ICR1 定义的，则 OC1A 或 ICF1 标志将与 TOV1 在同一个时钟周期置位。如果中断使能，则可以在中断服务程序里来更新 TOP 以及比较数据。

改变 TOP 值时必须保证新的 TOP 值不小于所有比较寄存器的数值，否则 TCNT1 与 OCR1x 不会出现比较匹配。使用固定的 TOP 值，向任意 OCR1x 寄存器写入数据时未使用的位将屏蔽为“0”。

定义 TOP 值时更新 ICR1 与 OCR1A 的步骤需要注意 ICR1 寄存器不是双缓冲寄存器，这意味着当计数器以无预分频器或很低的预分频工作时，给 ICR1 赋予一个小的数值时存在新写入的 ICR1 数值比 TCNT1 当前值小的危险，结果是计数器将丢失一次比较匹配。在下一次比较匹配发生之前，计数器不得不先计数到最大值 0xFFFF，然后再从 0x0000 开始计数，直到比较匹配出现，而 OCR1A 寄存器则是双缓冲寄存器。这一特性决定 OCR1A 可以随时写入，写入的数据被放入 OCR1A 缓冲寄存器。在 TCNT1 与 TOP 匹配后的下一个时钟周期，

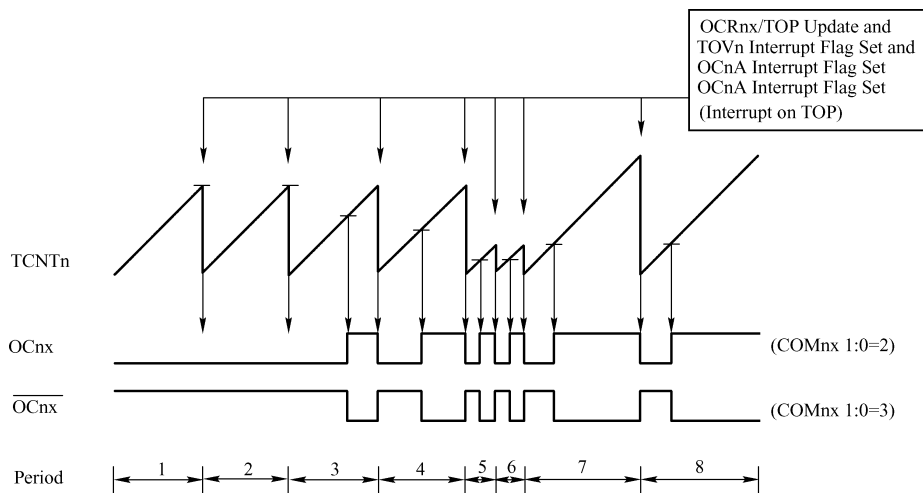


图 6.5 快速 PWM 工作模式的时序图

OCR1A 比较寄存器的内容被缓冲寄存器的数据所更新。在同一个时钟周期 TCNT1 被清零，而 TOV1 标志被设置。

使用固定 TOP 值时最好使用 ICR1 寄存器定义 TOP。这样 OCR1A 就可以用于在 OC1A 输出 PWM 波形。但是如果 PWM 基频不断变化（通过改变 TOP 值），OCR1A 的双缓冲特性使其更适合于这个应用。

当 T/C1 工作于快速 PWM 工作模式时，比较单元可以在 OC1x 引脚上输出 PWM 波形。设置 COM11 ~ COM10 为 2 可以产生普通的 PWM 信号；为 3 则可以产生反向 PWM 波形。此外，如果要真正从物理引脚上输出信号，还必须将 OC1x 的数据方向 DDR\_OC1x 设置为输出。产生 PWM 波形的机理是 OC1x 寄存器在 OCR1x 与 TCNT1 匹配时置位或者清零，以及在计数器清零（从 TOP 变为 BOTTOM）的那一个定时器时钟周期清零（或置位）。

输出的 PWM 频率可以通过如下公式计算得到：

$$f_{\text{OCnxPWM}} = \frac{f_{\text{clkIO}}}{N \times (1 + \text{TOP})}$$

其中，变量  $N$  代表分频因子（1、8、64、256 或 1024）。

OCR1x 寄存器为极限值表示快速 PWM 模式的一些特殊情况，若 OCR1x 等于 BOTTOM (0x0000)，输出为出现在第 TOP + 1 个定时器时钟周期的窄脉冲；OCR1x 为 TOP 时，根据 COM11 ~ COM10 的设定，输出恒为高电平或低电平。

通过设定 OC1A 在比较匹配时进行逻辑电平取反（COM1A1 ~ COM1A10 = 1），可以得到占空比为 50% 的周期信号，但是这只适用于 OCR1A 用来定义 TOP 值的情况。OCR1A 为 0 (0x0000) 时信号有最高频率  $f_{\text{OC1A}} = f_{\text{clkIO}}/2$ 。这个特性类似于 CTC 工作模式下的 OC1A 取反操作，不同之处是快速 PWM 模式具有双缓冲。

#### 4. 相位修正 PWM 工作模式

相位修正 PWM 工作模式为用户提供了一个获得高精度的、准确相位 PWM 波形的方法。与相位和频率修正工作模式类似，此工作模式基于双斜坡操作，计数器重复地从 BOTTOM

计到 TOP，然后又从 TOP 倒退回到 BOTTOM。在一般的比较输出工作模式下，当计数器往 TOP 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将清零为低电平；而在计数器往 BOTTOM 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将置位为高电平，工作于反向比较输出时则正好相反。与单斜坡操作相比，双斜坡操作可获得的最大频率要小，但其具有对称特性。相位修正 PWM 工作模式的 PWM 分辨率固定为 8、9 或 10 位，或由 ICR1 或 OCR1A 来定义。最小分辨率为 2 bit（ICR1 或 OCR1A 设为 0x0003），最大分辨率为 16 位（ICR1 或 OCR1A 设为 MAX）。PWM 分辨率位数可用下式计算：

$$R_{PCPWM} = \frac{\log(TOP + 1)}{\log(2)}$$

当 T/C1 工作于相位修正 PWM 工作模式时，计数器的数值一直累加到固定值 0x00FF、0x01FF、0x03FF、ICR1 或 OCR1A，然后改变计数方向，在同一个定时器时钟周期里 TCNT1 值等于 TOP 值。其具体的时序图如图 6.6 所示。图中给出了当使用 OCR1A 或 ICR1 来定义 TOP 值时的相位修正 PWM 工作模式。图中柱状的 TCNTn 表示双边斜坡操作。方框图同时包含了普通的 PWM 输出以及反向 PWM 输出。TCNT1 斜坡上的短水平线表示 OCR1x 和 TCNT1 的匹配比较，比较匹配后 OC1x 中断标志置位。

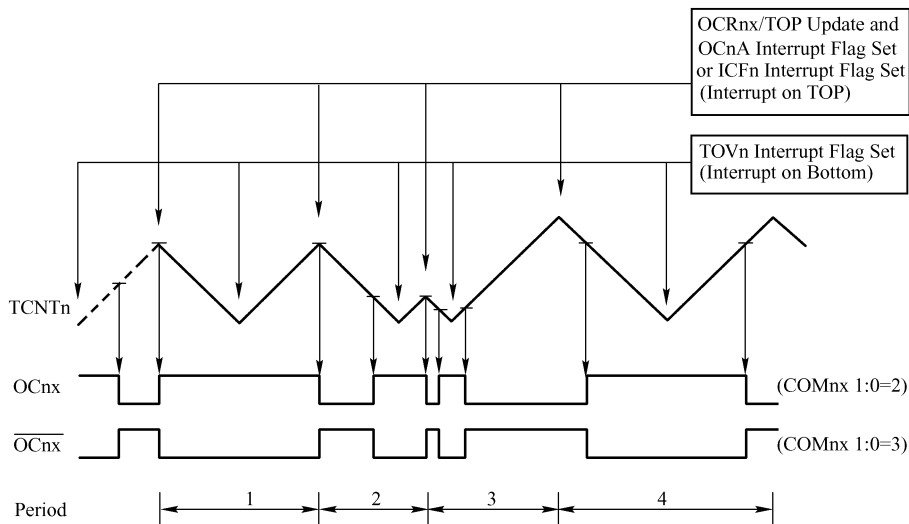


图 6.6 相位修正 PWM 工作模式的时序图

计数器数值达到 BOTTOM 时，T/C1 溢出标志 TOV1 被置位，若 TOP 由 OCR1A 或 ICR1 定义，在 OCR1x 寄存器通过双缓冲方式得到更新的同一个时钟周期里 OC1A 或 ICF1 标志被置位，并且产生一个中断事件。

改变 TOP 值时必须保证新的 TOP 值不小于所有比较寄存器的数值，否则 TCNT1 与 OCR1x 不会出现比较匹配。使用固定的 TOP 值时，向任意 OCR1x 寄存器写入数据时未使用的位将屏蔽为“0”。在图 6.6 所示的第 3 个周期中，在 T/C1 运行于相位修正模式时改变 TOP 值导致了不对称输出，其原因在于 OCR1x 寄存器的更新时间。由于 OCR1x 的更新时刻为定时器/计数器达到 TOP 值时，因此 PWM 的循环周期起始于此，也终止于此，也就是说，

下降斜坡的长度取决于上一个 TOP 值,而上升斜坡的长度取决于新的 TOP 值。若这两个值不同,一个周期内两个斜坡长度不同,输出也就不对称了。若要在 T/C1 运行时改变 TOP 值,最好用相位与频率修正工作模式代替相位修正工作模式。若 TOP 值保持不变,那么这两种工作模式实际没有区别。

工作于相位修正 PWM 工作模式时,比较单元可以在 OC1x 引脚输出 PWM 波形。设置 COM11 ~ COM10 为 2 可以产生普通的 PWM,设置 COM11 ~ COM10 为 3 可以产生反向 PWM。要真正从物理引脚上输出信号,还必须将 OC1x 的数据方向 DDR\_OC1x 设置为输出。OCR1x 和 TCNT1 比较匹配发生时,OC1x 寄存器将产生相应的清零或置位操作,从而产生 PWM 波形。工作于相位修正模式时,PWM 频率可由如下公式获得:

$$f_{\text{OCnxPCPWM}} = \frac{f_{\text{clkIO}}}{2 \times N \times \text{TOP}}$$

其中,变量  $N$  表示预分频因子 (1、8、64、256 或 1024)。

OCR1x 寄存器处于极值时表示相位修正 PWM 工作模式的一些特殊情况。在普通 PWM 工作模式下,若 OCR1x 等于 BOTTOM,输出一直保持为低电平;若 OCR1x 等于 TOP,输出则保持为高电平。反向 PWM 模式正好相反,如果 OCR1A 用来定义 TOP 值,且 COM11 ~ COM10 = 1,OC1A 输出占空比为 50% 的周期信号。

### 5. 相位与频率修正 PWM 工作模式

相频修正 PWM 工作模式可以产生高精度的、相位与频率都准确的 PWM 波形。与相位修正工作模式类似,相频修正 PWM 工作模式是基于双斜坡操作的,计时器重复地从 BOTTOM 计到 TOP,然后又从 TOP 倒退回到 BOTTOM。在一般的比较输出工作模式下,当计时器往 TOP 计数时若 TCNT1 与 OCR1x 匹配,OC1x 将清零为低电平;而在计时器往 BOTTOM 计数时 TCNT1 与 OCR1x 匹配,OC1x 将置位为高电平,工作于反向输出比较时则正好相反。与单斜坡操作相比,双斜坡操作可获得的最大频率要小。

相频修正修正 PWM 工作模式与相位修正 PWM 工作模式的主要区别在于 OCR1x 寄存器的更新时间,如图 6.7 所示。

相频修正 PWM 工作模式的 PWM 分辨率可由 ICR1 或 OCR1A 定义。最小分辨率为 2 bit (ICR1 或 OCR1A 设为 0x0003),最大分辨率为 16 位 (ICR1 或 OCR1A 设为 MAX)。PWM 的分辨率位数可用以下公式计算:

$$R_{\text{PCPWM}} = \frac{\log(\text{TOP} + 1)}{\log(2)}$$

T/C1 处于相频修正 PWM 工作模式时,计数器的数值一直累加到 ICR1 或 OCR1A,然后改变计数方向,在同一个定时器时钟里 TCNT1 值等于 TOP 值,如图 6.7 所示。图中给出了当使用 OCR1A 或 ICR1 来定义 TOP 值时的相频修正 PWM 工作模式。图中柱状的 TCNT1 表示双边斜坡操作,方框图同时包含普通的 PWM 输出以及反向 PWM 输出。TCNT1 斜坡上的短水平线表示 OCR1x 和 TCNT1 的匹配比较。比较匹配发生时,OC1x 中断标志将被置位。

在 OCR1x 寄存器通过双缓冲方式得到更新的同一个时钟周期里,T/C1 溢出标志 TOV1 置位。若 TOP 由 OCR1A 或 ICR1 定义,则当 TCNT1 达到 TOP 值时,OC1A 或 CF1 置位。这



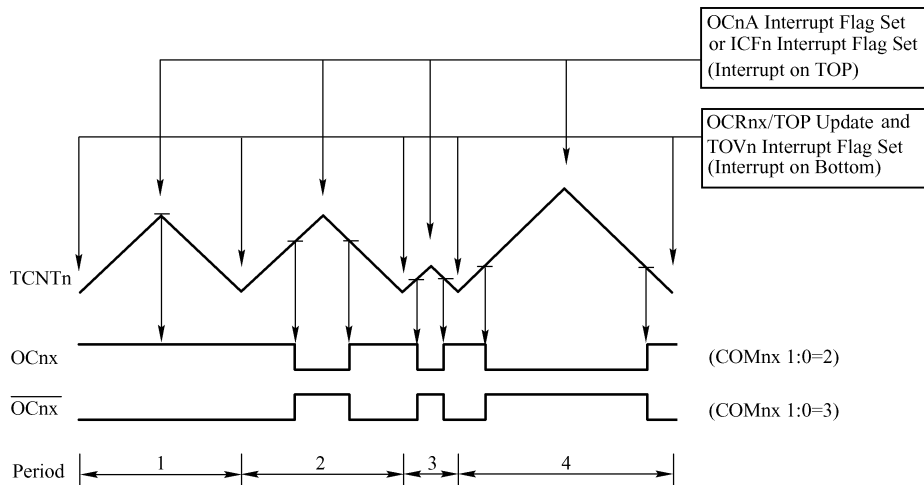


图 6.7 相位与频率修正 PWM 工作模式的时序图

些中断标志位可用在每次计数器达到 TOP 或 BOTTOM 时产生中断。改变 TOP 值时必须保证新的 TOP 值不小于所有比较寄存器的数值，否则 TCNT1 与 OCR1x 不会产生比较匹配。

如图 6.7 所示，与相位修正工作模式形成对照的是，相频修正 PWM 工作模式生成的输出在所有的周期中均为对称信号。这是由于 OCR1x 在 BOTTOM 得到更新，上升与下降斜坡长度始终相等。因此输出脉冲为对称的，确保了频率是正确的。使用固定 TOP 值时最好使用 ICR1 寄存器定义 TOP，这样 OCR1A 就可以用于在 OC1A 输出 PWM 波。但是如果 PWM 基频不断变化（通过改变 TOP 值），OCR1A 的双缓冲特性使其更适合于这个应用。

当 T/C1 工作于相频修正 PWM 工作模式时，比较单元可以在 OC1x 引脚上输出 PWM 波形。设置 COM11 ~ COM10 为 2 可以产生普通的 PWM 信号；为 3 则可以产生反向 PWM 波形。要想真正输出信号还必须将 OC1x 的数据方向设置为输出，产生 PWM 波形的机理是 OC1x 寄存器在 OCR1x 与升序计数的 TCNT1 匹配时置位（或清零），与降序计数的 TCNT1 匹配时清零（或置位）。输出的 PWM 频率可以通过如下公式计算得到：

$$f_{\text{OCnxPFCPWM}} = \frac{f_{\text{clk10}}}{2 \times N \times \text{TOP}}$$

其中，变量  $N$  代表分频因子（1、8、64、256 或 1024）。

OCR1x 寄存器处于极值时说明了相频修正 PWM 工作模式的一些特殊情况，在普通 PWM 工作模式下，若 OCR1x 等于 BOTTOM，输出一直保持为低电平；若 OCR1x 等于 TOP，则输出保持为高电平。反向 PWM 模式则正好相反。如果 OCR1A 用来定义 TOP 且 COM11 ~ COM10 = 1，则 OC1A 输出占空比为 50% 的周期信号。

### 6.3 定时计数器 T/C2

ATmega16 的定时计数器 T/C2 是一个通用单通道 8 位定时计数器，它支持自动重载的比较匹配时清零定时器，支持无干扰脉冲、相位正确的脉宽调制器 PWM，可以作为频率发生器、10 位时钟预分频器，并且允许使用外部的 32 kHz 晶体作为独立的时钟源。



### 6.3.1 T/C2 的相关寄存器

T/C2 相关的控制寄存器有 T/C2 控制寄存器 TCCR2，T/C2 定时计数寄存器 TCNT2，T/C2 输出比较寄存器 OCR2 等，相关说明如下。

#### 1. T/C2 控制寄存器 TCCR2

TCCR2 寄存器用于对 T/C2 进行相关控制，其内部结构如表 6.24 所示。

表 6.24 TCCR2 寄存器内部结构

BIT	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20
读/写	R/W	R/W	R/W	R/W	W	W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- FOC2：强制输出比较位。该位仅在 WGM 位指明 T/C2 工作于非 PWM 模式下时才有效，但是在 PWM 工作模式下，对 TCCR2 的写操作要对其清零，当该位被置“1”后，波形发生器将立即进行比较操作，比较匹配输出引脚 OC2 将按照 COM21 ~ COM20 的设置输出相应的电平。需要注意的是，FOC2 类似一个锁存信号，真正对强制输出比较起作用的是 COM21 ~ COM20 的设置。FOC2 位不会引发任何中断，也不会在使用 OCR2 作为 TOP 的 CTC 工作模式下对定时器进行清零，对 FOC2 读操作的返回值永远为 0。
- WGM21 ~ WGM20：用于控制 T/C2 的工作模式，如表 6.25 所示。

表 6.25 波形产生模式的位定义

模式	WGM21 (CTC2)	WGM20 (PWM2)	T/C2 的工作模式	TOP	OCR2 的 更新时间	TOV2 的 置位时间
0	0	0	普通	0xFF	立即更新	MAX
1	0	1	相位修正 PWM	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR2	立即更新	MAX
3	1	1	快速 PWM	0xFF	TOP	MAX

- COM21 ~ COM20：比较匹配输出模式控制位，用于控制比较匹配发生时输出引脚 OC0 的电平。如果 COM21 ~ COM20 中的一位或全部都置位，OC2 以比较匹配输出的方式进行工作，同时其引脚方向控制位要设置为“1”，以使能输出驱动。当 OC2 连接到物理引脚上时，COM21 ~ COM20 的功能依赖于 WGM21 ~ WGM20 的设置，表 6.26 是其对应的设置方式。

表 6.26 比较输出模式，非 PWM 模式

COM21	COM20	说明
0	0	正常的端口操作，OC2 未连接
0	1	比较匹配时 OC2 取反
1	0	比较匹配时 OC2 清零
1	1	比较匹配时 OC2 置位





表 6.27 是当 WGM21 ~ WGM20 设置为快速 PWM 工作模式时, COM21 ~ COM20 对应的功能。

表 6.27 比较输出模式, 快速 PWM 模式

COM21	COM20	说 明
0	0	正常的端口操作, OC2 未连接
0	1	保留
1	0	比较匹配时 OC2 清零, 计数到 TOP 时 OC2 置位
1	1	比较匹配时 OC2 置位, 计数到 TOP 时 OC2 清零

表 6.28 是当 WGM20 ~ WGM20 设置为相位修正 PWM 工作模式时, COM21 ~ COM20 对应的功能。

表 6.28 比较输出模式, 相位修正 PWM 模式

COM21	COM20	说 明
0	0	正常的端口操作, OC2 未连接
0	1	保留
1	0	在升序计数时发生比较匹配清零 OC2, 降序计数时发生比较匹配将置位 OC2
1	1	在升序计数时发生比较匹配置位 OC2, 降序计数时发生比较匹配将清零 OC2

- CS22 ~ CS20: T/C2 的时钟源选择位, 其说明如表 6.29 所示。

表 6.29 时钟选择

CS22	CS21	CS20	说 明
0	0	0	无时钟, T/C2 不工作
0	0	1	CLK <sub>T2</sub>
0	1	0	CLK <sub>T2</sub> /8
0	1	1	CLK <sub>T2</sub> /32
1	0	0	CLK <sub>T2</sub> /64
1	0	1	CLK <sub>T2</sub> /128
1	1	0	CLK <sub>T2</sub> /256
1	1	1	CLK <sub>T2</sub> /1024

## 2. T/C2 定时计数器寄存器 TCNT2

T/C2 的定时计数器寄存器 TCNT2 用于 T/C2 的定时计数, 其内部结构如表 6.30 所示。可以对 T/C2 寄存器 TCNT2 的 8 位数据进行读/写访问, 对 TCNT2 寄存器的写访问将在下一个时钟阻止比较匹配, 在计数器运行的过程中修改 TCNT2 的数值有可能丢失一次 TCNT2 和 OCR2 的比较匹配。

表 6.30 TCNT2 寄存器

BIT	TCNT2[7~0]							
读/写	R/W	R/W	R/W	R/W	W	W	R/W	R/W
初始值	0	0	0	0	0	0	0	0



### 3. T/C2 输出比较寄存器 OCR2

T/C2 输出比较寄存器包含一个 8 位的数据，不间断地与计数器数值 TCNT2 进行比较。匹配事件可以用来产生输出比较中断，或者用来在 OC2 引脚上产生波形，其内部结构如表 6.31 所示。

表 6.31 OCR2 寄存器

BIT	OCR2[7~0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

### 4. T/C2 中断屏蔽寄存器 TIMSK

TIMSK 寄存器用于对 T/C2 的中断进行控制，其内部结构如表 6.32 所示。

表 6.32 TIMSK 寄存器内部结构

BIT	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- OCIE2: T/C2 输出比较匹配中断使能位。当 OCIE2 和状态寄存器的全局中断使能位 I 都被置“1”时，T/C2 的输出比较匹配 A 中断使能；当有 T/C2 的比较匹配发生，即 TIFR 中的 OCF2 置位时，进入中断服务程序。
- TOIE2: T/C2 溢出中断使能位。当 TOIE2 和状态寄存器的全局中断使能位 I 都被置“1”时，T/C2 的溢出中断被使能；当 T/C2 发生溢出，即 TIFR 中的 TOV2 位置位时，进入中断服务程序。

### 5. T/C2 中断标志寄存器 TIFR

TIFR 寄存器用于标志 T/C2 的中断状态，其内部结构如表 6.33 所示。

表 6.33 TIMSK 寄存器内部结构

BIT	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- OCF2: 输出比较标志位 2。当 T/C2 与 OCR2（输出比较寄存器 2）的值匹配时，OCF2 被置位，在中断服务子程序里被硬件清零，也可以通过对其写入“1”来清零；当 SREG 中的位 I、OCIE2 和 OCF2 都被置位时，进入中断服务子程序。
- TOV2: T/C2 溢出标志位。当 T/C2 溢出时，TOV2 被置位，进入相应的中断服务程序，此位被硬件清零，TOV2 也可以通过写“1”来清零；当 SREG 中的位 I、TOIE2 和 TOV2 都被置位时，进入中断服务程序，在 PWM 工作模式中，当 T/C2 在 0x00 时改变计数方向，同时 TOV2 置位。



### 6.3.2 T/C2 的工作模式

T/C2 定时计数器有普通工作模式、比较匹配时清除定时器 (CTC) 工作模式、快速 PWM 工作模式、相位修正 PWM 工作模式。

T/C2 和输出比较引脚的行为由波形发生模式 (WGM21 ~ WGM20) 及比较输出模式 (COM21 ~ COM20) 的控制位决定, 比较输出模式对计数序列没有影响, 而波形产生模式对计数序列则有影响。COM21 ~ COM20 用于控制 PWM 的输出是否反极性, 而在非 PWM 工作模式下 COM21 ~ COM20 用于控制输出是否应该在比较匹配发生时置位、清零, 或是电平取反。

#### 1. 普通工作模式

当 WGM21 ~ WGM20 = 0 时, T/C2 工作于普通工作模式, 这是最简单的工作模式, 在此工作模式下计数器不停地累加, 当计数达到 8bit 的最大值后 ( $TOP = 0xFF$ ), 由于数值溢出, T/C2 简单地返回到最小值  $0x00$  重新开始计数。在 TCNT0 变为零的同一个定时器时钟里 T/C2 的溢出标志 TOV2 置位, 此时 TOV2 类似第 9 位, 只是只能置位, 不会清零, 但由于定时器中断服务程序能够自动清零 TOV2, 因此可以通过软件提高定时器的分辨率。在普通工作模式下没有什么需要特殊考虑的, 用户可以随时写入新的计数器数值。

#### 2. 比较匹配清除定时器 (CTC) 工作模式

在 T/C2 的 CTC 工作模式 (WGM21 ~ WGM20 = 2) 下, OCR2 寄存器用于调节计数器的分辨率, 当计数器的数值 TCNT2 等于 OCR2 时, 计数器清零。OCR2 定义了计数器的 TOP 值, 也即计数器的分辨率, 在此工作模式下可以很容易地控制比较匹配输出的频率, 也简化了外部事件计数的操作。

CTC 工作模式的时序图如图 6.8 所示, 计数器数值 TCNT2 一直累加到 TCNT2 与 OCR2 匹配, 然后 TCNT2 清零。

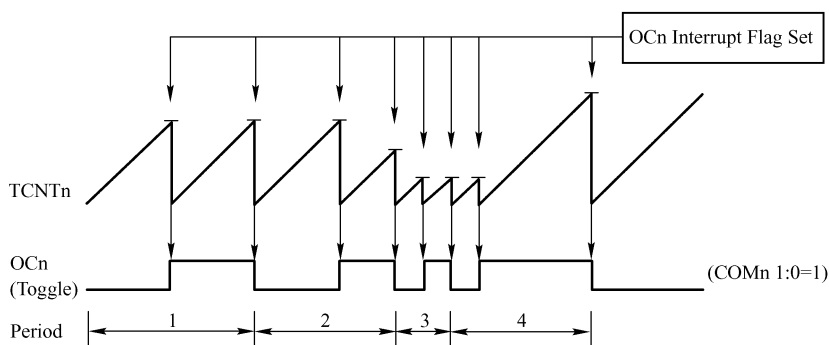


图 6.8 CTC 工作模式的时序图

利用 OCF2 标志可以在计数器数值达到 TOP 即产生中断, 在中断服务程序里可以更新 TOP 的数值。由于 CTC 工作模式下没有双缓冲功能, 在计数器以无预分频器或很低的预分频器工作时, 将 TOP 更改为接近 BOTTOM 的数值时要注意, 如果写入 OCR2 的数值小于当



前 TCNT2 的数值，计数器将丢失一次比较匹配。在下一次比较匹配发生之前，计数器不得不先计数到最大值 0xFF，然后再从 0x00 开始计数到 OCR2。

为了在 CTC 工作模式下得到波形输出，可以设置 OC2 在每次比较匹配发生时改变逻辑电平。这可以通过设置 COM21 ~ COM20 = 1 来完成，在期望获得 OC2 输出之前，首先要将其端口设置为输出。波形发生器能够产生的最大频率为：

$$f_{OC2} = \frac{f_{clk\_lo}}{2}$$

其频率由如下公式确定：

$$f_{OCn} = \frac{f_{clkIO}}{2 \times N \times (1 + OCRn)}$$

其中，变量  $N$  代表预分频因子（1、8、32、64、128、256 或 1024）。

### 3. 快速 PWM 工作模式

当 WGM21 ~ WGM20 = 3 时，ATmega16 工作于快速 PWM 工作模式，它可用来产生高频的 PWM 波形。快速 PWM 模式与其他 PWM 工作模式的不同之处是其单边斜坡工作方式，计数器从 BOTTOM 计到 MAX，然后立即回到 BOTTOM 重新开始。对于普通的比较输出模式，输出比较引脚 OC2 在 TCNT2 与 OCR2 匹配时清零，在 BOTTOM 时置位；对于反向比较输出模式，OC2 的动作正好相反。由于使用了单边斜坡模式，快速 PWM 模式的工作频率比使用双斜坡的相位修正 PWM 工作模式高一倍。此高频操作特性使得快速 PWM 工作模式十分适合于功率调节、整流和 DAC 应用。高频可以减小外部元器件（电感、电容）的物理尺寸，从而降低系统成本。当 T/C2 工作于快速 PWM 工作模式时，计数器的数值一直增加到 MAX，然后在后面的一个时钟周期清零，其具体的时序图如图 6.9 所示。图中柱状的 TCNT0 表示单边斜坡操作，方框图同时包含了普通的 PWM 输出以及方向 PWM 输出，TCNT2 斜坡上的短水平线表示 OCR2 和 TCNT2 的比较匹配。

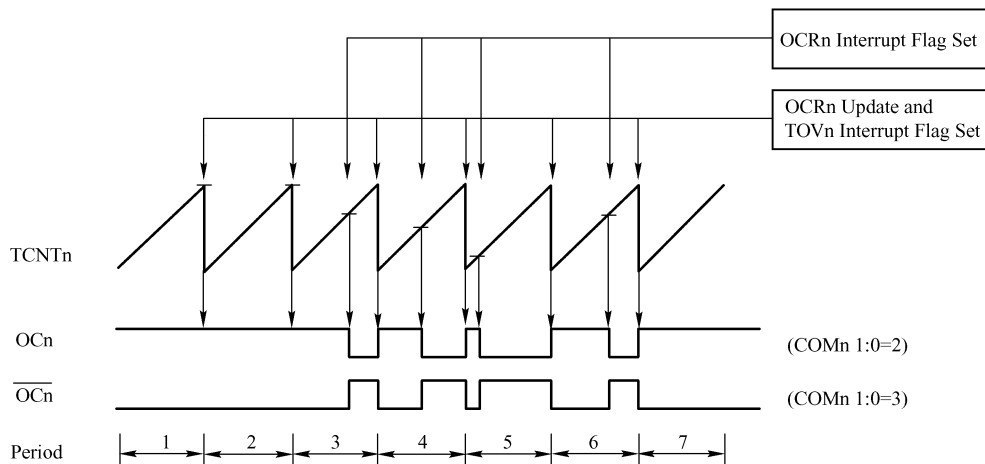


图 6.9 快速 PWM 工作模式的时序图

当 T/C2 计时器数值达到 MAX 时，T/C 溢出标志 TOV2 置位，如果中断使能，在中断服



务程序里可以更新比较值。

当 T/C2 工作于快速 PWM 工作模式时, 比较单元可以在 OC2 引脚上输出 PWM 波形, 当 COM21 ~ COM20 = 2 时可以产生普通的 PWM 信号, 为 3 时则可以产生反向 PWM 波形, 但是如果想在引脚上得到输出信号还必须将 OC2 的数据方向设置为输出。T/C2 产生 PWM 波形的机理是 OC2 寄存器在 OCR2 与 TCNT2 匹配时置位或清零, 以及在计数器清零 (从 MAX 变为 BOTTOM) 的那一个定时器时钟周期清零或置位。输出的 PWM 频率可以通过如下公式计算得到:

$$f_{\text{ocnPWM}} = \frac{f_{\text{clkIO}}}{N \times 256}$$

其中, 变量  $N$  代表分频因子 (1、8、32、64、128、256 或 1024)。

OCR2 寄存器为极限值时表示快速 PWM 工作模式的一些特殊情况, 若 OCR2 等于 BOTTOM, 输出为出现在第 MAX + 1 个定时器时钟周期的窄脉冲, 当 OCR2 为 MAX 时, 根据 COM21 ~ COM20 的设定, 输出恒为高电平或低电平。

通过设定 OC2 在比较匹配时进行逻辑电平取反 (COM21 ~ COM20 = 1), 可以得到占空比为 50% 的周期信号, 当 OCR2 为 0 时信号有最高频率:

$$f_{\text{oc2}} = \frac{f_{\text{clkIo}}}{2}$$

这个特性类似于 CTC 工作模式下的 OC2 取反操作, 不同之处在于快速 PWM 工作模式下具有双缓冲。

#### 4. 相位修正 PWM 工作模式

当 WGM21 ~ WGM20 = 1 时, T/C2 工作于相位修正 PWM 工作模式, 该模式为用户提供了一个获得高精度相位修正 PWM 波形的办法。此模式基于双斜坡操作, 计时器重复地从 BOTTOM 计到 MAX, 然后又从 MAX 倒退回到 BOTTOM。在一般的比较输出模式下, 当计时器往 MAX 计数时, 若 TCNT2 与 OCR2 匹配, OC2 将清零为低电平; 而在计时器往 BOTTOM 计数时, 若 TCNT2 与 OCR2 匹配, OC2 将置位为高电平, 工作于反向输出比较时则正好相反。与单斜坡操作相比, 双斜坡操作可获得的最大频率要小。但由于其对称的特性, 十分适合于电机控制。

相位修正 PWM 工作模式的 PWM 精度固定为 8bit, 计时器不断地累加直到 MAX, 然后开始减计数, 在一个定时器时钟周期里 TCNT2 的值等于 MAX, 相位修正 PWM 工作模式的时序图如图 6.10 所示。

在图中, TCNT2 的数值用柱状图表示, 以说明双斜坡操作, 该图同时说明了普通 PWM 的输出和反向 PWM 的输出。TCNT2 斜坡上的短水平线表示 OCR2 和 TCNT2 的比较匹配。当计时器达到 BOTTOM 时, T/C2 溢出标志位 TOV2 被置位。此标志位可用来产生中断。

当 ATmega16 工作于相位修正 PWM 工作模式时, 其比较单元可以在 OC2 引脚产生 PWM 波形, 在 COM21 ~ COM20 设置为 “2” 时则产生普通相位的 PWM, 设置 COM21 ~ COM20 为 3 则产生反向 PWM 信号, 但是需要注意的是, 要想在引脚上得到输出信号还必须将 OC2 的数据方向设置为输出。当 OCR2 和 TCNT2 比较匹配发生时, OC2 寄存器将产生相应的清零或置位操作, 从而产生 PWM 波形, 工作于相位修正模式时 PWM 频率可由下式公式获得:





$$f_{\text{ocnppwm}} = \frac{f_{\text{clklo}}}{N \times 510}$$

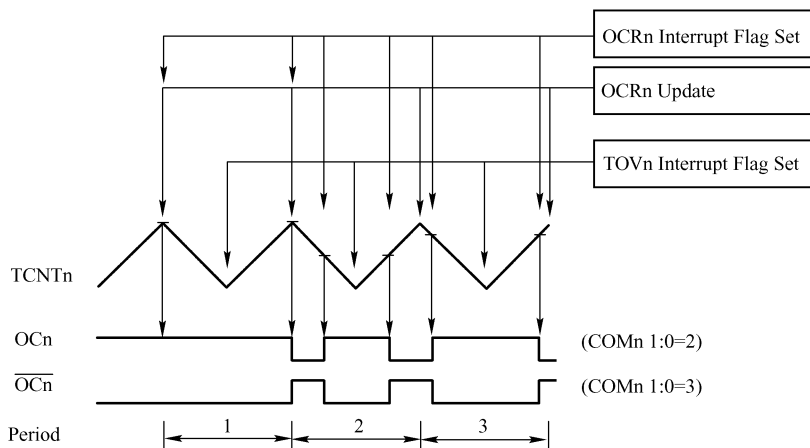


图 6.10 相位修正 PWM 工作模式的时序图



### 注意

变量  $N$  表示预分频因子（1、8、32、64、128、256 或 1024）。

OCR2 寄存器为极限值时表示相位修正 PWM 工作模式的一些特殊情况。在普通 PWM 工作模式下，若 OCR2 等于 BOTTOM，输出一直保持为低电平；若 OCR2 等于 MAX，则输出保持为高电平，在反向 PWM 工作模式下则正好相反。

从图 6.10 可以看到，在第 2 个周期，虽然没有发生比较匹配，OC2 也出现了一个从高到低的跳变，其目的是保证波形在 BOTTOM 两侧的对称。没有比较匹配时有如下两种情况会出现跳变。

- OCR2A 的值从 MAX 改变为其他数据，当 OCR2A 值为 MAX 时，引脚 OC2 的输出应该与前面降序计数比较匹配的结果相同。为保证波形在 BOTTOM 两侧对称，当 T/C2 的数值为 MAX 时，引脚 OC2 的输出又必须符合后面升序计数比较匹配的结果，此时就出现了虽然没有比较匹配发生，OC2 却仍然有跳变的现象。
- 定时器从一个比 OCR2A 大的值开始计数，并由此丢失了一次比较匹配，因此引入了没有比较匹配发生，OCn 却仍然有跳变的现象。

## 6.4 ATmega16 的定时计数器的应用实例

### 6.4.1 T/C0 控制 I/O 引脚输出方波

本应用是一个使用定时器 T/C0 控制 I/O 引脚输出方波的实例。

#### 1. 实例的设计思路

使 T/C0 工作于普通工作模式下，定时 20ms，然后在定时器溢出的中断服务子函数中将

I/O 引脚翻转即可。

## 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 6.11 所示，一个虚拟示波器连接到 ATmega16 的 PB0 引脚上，表 6.34 为实例中使用的 Proteus 器件。

表 6.34 应用实例器件列表

器件名称	大类库	子类库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体

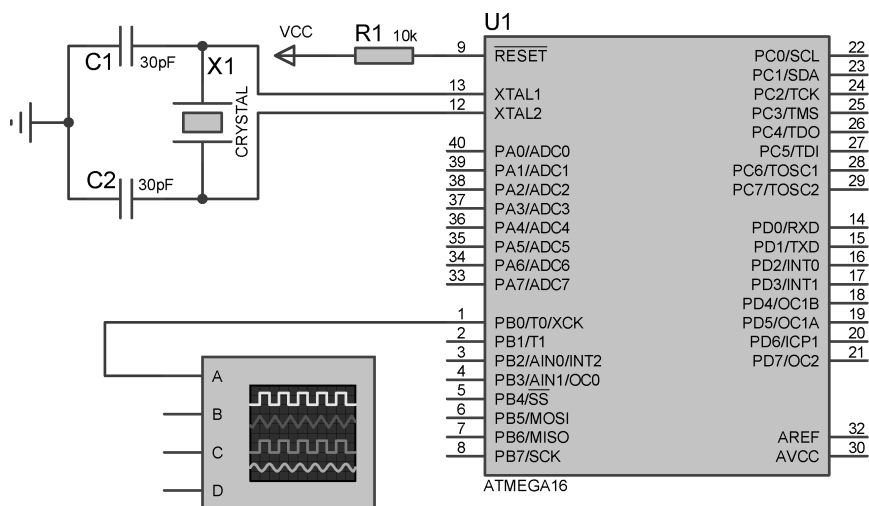


图 6.11 实例的 Proteus 电路

## 3. 实例的应用代码

实例的应用代码如例 6.1 所示。代码首先对 T/C0 的工作方式和 I/O 端口进行初始化，然后在 T/C0 的中断服务子函数中将 I/O 引脚上的电平翻转即可。

### 【例 6.1】定时器 T/C0 控制 I/O 引脚输出方波

```
#include <iom16v.h>
#include <macros.h>
//端口初始化
void port_init(void)
{
    PORTA = 0x00;
    DDRA = 0x00;
```

```

PORTB = 0x00;
DDRB = 0x01;                                //PB0 为输出
PORTC = 0x00;
DDRC = 0x00;
PORTD = 0x00;
DDRD = 0x00;
}
//T/C0 初始化子函数, 1024 分频, 定时 20ms
void timer0_init( void)
{
    TCCR0 = 0x00;
    TCNT0 = 0x64;                            //设置 TCNT 初始值
    OCR0 = 0x9C;
    TCCR0 = 0x05;
}
//TC0 溢出中断服务子程序
#pragma interrupt_handler timer0_ovf_isr:iv_TIM0_OVF
void timer0_ovf_isr( void)
{
    TCNT0 = 0x64;                            //重装初始值
    PORTB ^= BIT(0);                         //翻转
}
//ATmega16 初始化
void init_devices( void)
{
    CLI();
    port_init();
    timer0_init();
    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x01;
    SEI();
}
//主函数
void main( void)
{
    init_devices();
    while( 1)
    {
    }
}

```





#### 4. 实例的仿真结果和说明

点击运行，调整示波器，可以在 PB0 引脚上观察到相应的方波输出，如图 6.12 所示。

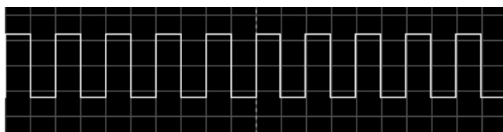


图 6.12 实例的仿真运行结果



#### 注意

读者可以自行修改 T/C0 的定时长度来修改方波的频率和宽度。

### 6.4.2 T/C1 控制 I/O 引脚输出 PWM

ATmega16 的定时计数器可以控制其 I/O 引脚产生相应的脉冲调制宽度 (PWM) 信号，本应用是使用定时计数器产生一个 PWM 波形的实例。

脉冲宽度调制 (Pulse Width Modulation, PWM)，是一种改变方波的占空比以输出不同波形的方波，改变占空比的方法有三种。

(1) 定宽调频法：该方法是不改变高电平的维持时间，仅改变低电平的维持时间，这样调制电压频率也随之改变。

(2) 调宽调频法：该方法要求不改变低电平的维持时间，仅改变高电平的维持时间，这样调制电压频率也被改变。

(3) 定频调宽法：该方法是同时改变高低电平的维持时间，而两个维持时间的总和不变，即调制电压频率不变。

#### 1. 实例的设计思路

由于 T/C1 有快速 PWM 工作模式，在该工作模式下，当累加计数器 TCNT1 的值和 OCR1A 寄存器的值相等时，OC1A 引脚上的波形翻转，所以可以使用该方法来产生 PWM 波形。

#### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 6.11 所示，略有不同的是，PWM 波形是从 PD5 (OC1A) 引脚上输出的。

#### 3. 实例的应用代码

实例的应用代码如例 6.2 所示。代码对 T/C1 进行初始化，使得其工作在快速 PWM 工作模式下，此时即可在对应的引脚上观察到对应的 PWM 波形。

#### 【例 6.2】定时器产生 PWM 波形

```
#include <iom16v.h>
#include <macros.h>
//端口初始化
```

```

void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0x00;
    PORTB = 0x00;
    DDRB = 0x00;
    PORTC = 0x00;
    DDRC = 0x00;
    PORTD = 0x00;
    DDRD  = 0x20;                //PD5 为输出
}

//T/C1 的 PWM 初始化, 快速 PWM 输出
void timer1_init( void)
{
    TCCR1B = 0x00;
    //设置初始化值, 决定输出的 PWM 波形的频率
    TCNT1H = 0xFC;
    TCNT1L = 0x01;
    OCR1AH = 0x03;
    OCR1AL = 0xFF;              //设置比较寄存器的初始化值
    OCR1BH = 0x03;
    OCR1BL = 0xFF;
    ICR1H = 0x03;
    ICR1L = 0xFF;
    TCCR1A = 0x43;
    TCCR1B = 0x02;              //启动 T/C1
}

//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    timer1_init();

    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主程序
void main( void)

```

```

{
    init_devices();
    while(1)
    {
        }
    }
}

```

#### 4. 实例的仿真结果和说明

在 PD5 (OC1A) 引脚上连接一个示波器, 点击运行, 调节示波器, 可以看到对应的波形输出, 如图 6.13 所示。

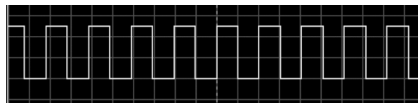


图 6.13 实例的仿真运行输出



#### 总结

可以通过调节 T/C1 的预置值来获得不同宽度和频率的 PWM 波形, 读者可以自行尝试, 由于此类 PWM 波形输出完全由 ATmega16 硬件完成, 对软件代码的要求极低, 所以占用的系统资源比较小, 且输出波形也更加精确。

### 6.4.3 外部晶体秒定时

ATmega16 的 T/C2 支持外接晶体作为时钟源, 本应用是一个使用 ATmega16 的 PC6 (TOSC2) 和 PC7 (TOSC1) 外接 32.768 kHz 钟表晶体作为时钟源来实现秒定时的实例。



#### 注意

32.768 kHz 是最常用的钟表晶体。

#### 1. 实例的设计思路

在初始化时, 将 T/C0 的时钟源选择为外部时钟源, 并且计算出相应的溢出参数, 在对应的 T/C0 中断服务子程序中将 I/O 引脚上的电平翻转即可。

#### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 6.14 所示, 晶体 X2 的频率设置为 32768Hz, 表 6.35 为实例所使用的 Proteus 器件列表。

表 6.35 应用实例器件列表

器件名称	大类库	子类库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	-	晶体

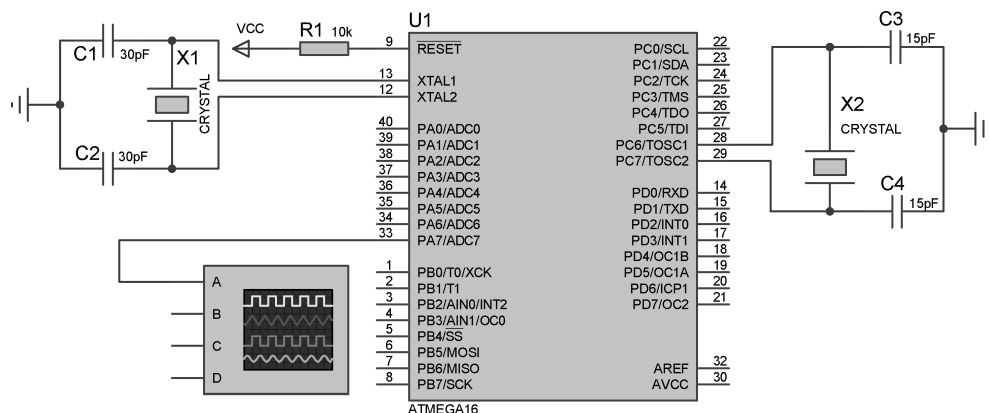


图 6.14 实例的 Proteus 电路

### 3. 实例的应用代码

实例的应用代码如例 6.3 所示，设置 T/C2 采用外部时钟源，512  $\mu$ s 溢出一次，在每次溢出时一个标志 secflg 的值进行翻转，当该标志位的值为 0x00 时，控制外部 I/O 引脚 PA7 上的电平翻转。

#### 【例 6.3】 定时计数器秒定时输出高低电平

```
#include <iom16v.h>
#include <macros.h>

unsigned char secflg = 0x00;           //秒定时标志
//端口初始化
void port_init( void )
{
    PORTA = 0x00;
    DDRA  = 0x80;                       //PA7 为输出
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
}
//T/C2 初始化
//普通工作模式
//采用外部时钟驱动
//512 $\mu$ s 溢出
void timer2_init( void )
{

```



```

TCCR2 = 0x00;           //停止
ASSR   = 0x08;          //使用外部时钟驱动
TCNT2 = 0x7D;           //设置初始化值
OCR2   = 0x83;
TCCR2 = 0x05;           //启动时钟
}

//T/C2 时钟中断
#pragma interrupt_handler timer2_ovf_isr:iv_TIM2_OVF
void timer2_ovf_isr( void)
{
    TCNT2 = 0x7D;        //重新设置初始化值
    if( secflg == 0x00)
    {
        secflg = 0xff;
    }
    else
    {
        secflg = 0x00;
        PORTA^ = BIT(7); //电平翻转
    }
}

//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    timer2_init();

    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x41;
    SEI();
}

//主程序
void main( void)
{
    init_devices();
    while(1)
    {
    }
}

```

#### 4. 实例的仿真结果和说明

在 PA7 引脚上放置一个示波器，点击运行，可以观察到相应的波形输出，如图 6.15 所示。

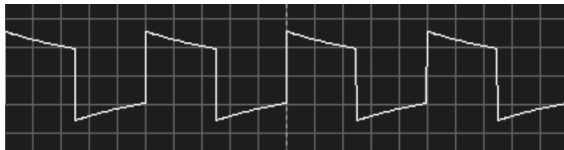


图 6.15 实例的仿真运行输出



#### 总结

图中波形的畸变可能是由于 Proteus 内置的虚拟示波器的扫描时间太长导致的。

## 第7章 ATmega16 单片机的串口

ATmega16 带有一个全双工的通用同步/异步串行收发模块 USART，它是一个功能丰富的串行通信接口，常用于多个 ATmega16 之间或者 ATmega16 和其他系统的数据通信，本章介绍该模块的详细使用方法。

### 7.1 ATmega16 串口的结构

ATmega16 串口模块的内部结构如图 7.1 所示，它可以分为时钟发生逻辑、数据发送逻辑和数据接收逻辑三个单元，其中控制寄存器由这三个单元共享，它们的详细说明如下。

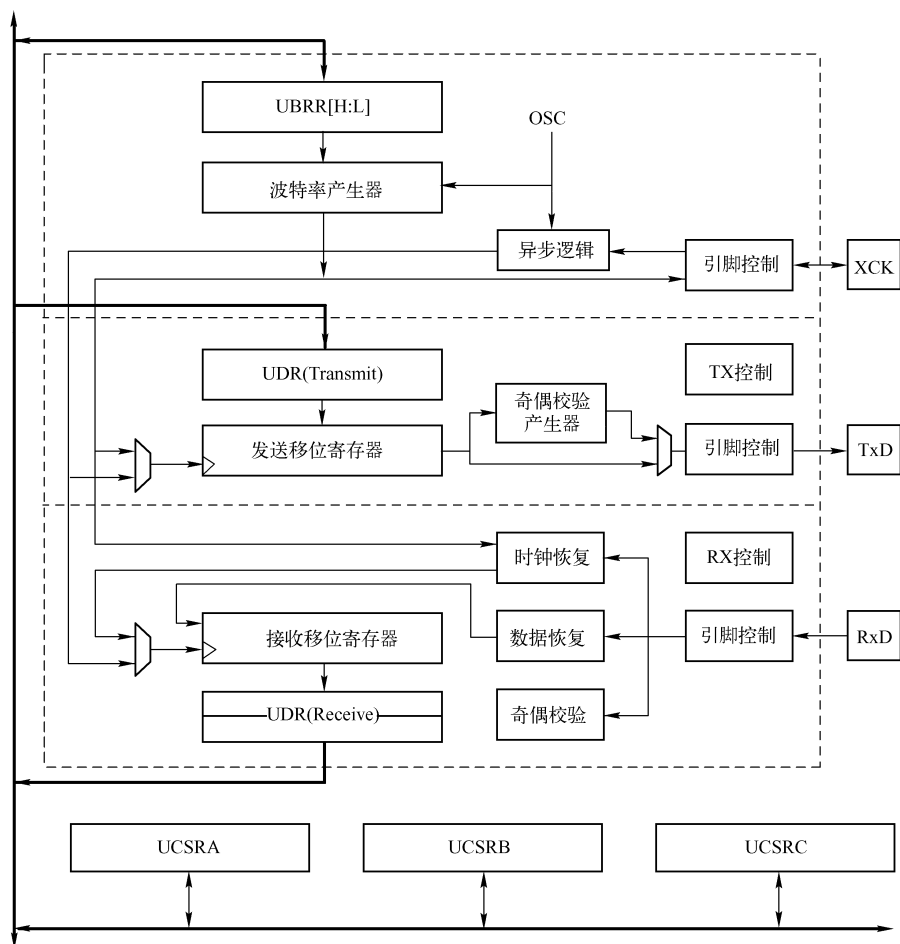


图 7.1 串口模块的内部结构



(1) 时钟发生逻辑模块。它包含同步逻辑和异步逻辑，用于给串口模块产生相关的驱动时序，其中 XCK（发送器时钟）引脚只用于同步传输模式。

(2) 数据发送逻辑模块。它由一个写缓冲器（UDR）、一个发送移位寄存器、奇偶发生器以及处理不同的帧格式所需的控制逻辑组成，写缓冲器可以保持连续发送数据而不会在数据帧之间引入延时。

(3) 数据接收逻辑模块。它包括时钟恢复单元和数据恢复单元，所以它是串口模块中最复杂的部分，其中恢复单元只能用于异步数据的接收。除了恢复单元，接收逻辑还包括奇偶校验、控制逻辑、一个接收移位寄存器和一个两级接收缓冲器（UDR）。数据接收器支持与发送器相同的帧格式，而且可以检测帧错误，数据过速和奇偶校验错误。

## 7.2 ATmega16 串口的寄存器

ATmega16 通过对相关寄存器的操作来完成对串口模块的控制，这些相关的寄存器包括串口数据寄存器（UDR）、串口控制和状态寄存器 A（UCSRA）、串口控制和状态寄存器 B（UCSRB）、串口控制和状态寄存器 C（UCSRC）和串口波特率寄存器（UBRRL 和 UBRRH）。

### 7.2.1 串口数据寄存器（UDR）

串口的发送数据缓冲寄存器（TXB）和接收数据缓冲寄存器（RXB）使用同一个 I/O 地址，如表 7.1 所示。将数据写入该寄存器时，实际操作的是发送数据缓冲寄存器（TXB），读 UDR 时，实际返回的是接收数据缓冲寄存器（RXB）的内容。

表 7.1 UDR 的两个寄存器

UDR 读	RXB[7~0]							
UDR 写	TXB[7~0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

### 7.2.2 串口控制和状态寄存器 A（UCSRA）

USART 控制和状态寄存器 A 主要用于存放串口的各种状态和对其进行部分控制，其内部结构如表 7.2 所示。

表 7.2 USART 控制和状态寄存器 A

BIT	7	6	6	4	3	2	1	0
位	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
读/写	R	R/W	R	R	R	R	R/W	R/W
初始值	0	0	1	0	0	0	0	0





- **RXC**: ATmega16 串口接收结束标志位。当接收缓冲器中有未读出的数据时, RXC 置“1”, 否则清“0”。接收器被禁止时, 接收缓冲器被刷新, RXC 清“0”。RXC 标志可用来产生接收结束中断 (参考 RXCIE 位的使用)。
- **TXC**: ATmega16 串口发送结束标志位。当发送移位缓冲器中的数据被发送完成, 且当发送缓冲器 (UDR) 为空时, TXC 置位。执行发送结束中断时, TXC 标志被自动清零, 也可以通过写“1”进行清除操作, TXC 标志可用来产生发送结束中断 (参考后面 TXCIE 位的使用)。
- **UDRE**: ATmega16 串口数据寄存器空标志。该位用于表示发送缓冲器 (UDR) 是否准备好接收新数据, 当 UDRE 为“1”时表明缓冲器为空, 已准备好进行数据接收。UDRE 标志可用来产生数据寄存器空中断 (参考 UDRIE 位的使用), 复位后 UDRE 被置“1”, 表明发送器已经就绪。
- **FE**: 帧错误标志位。如果接收缓冲器接收到的下一个字符有帧错误, 即接收缓冲器中的下一个字符的第一个停止位为“0”, 那么 FE 被置“1”, 该标志位一直有效直到接收缓冲器 (UDR) 被读取。当接收到的停止位为“1”时, FE 标志为“0”, 当对 UCSRA 进行写入操作时, 这一位要置“0”。
- **DOR**: 数据溢出标志位。当数据溢出时, DOR 被置“1”; 当接收缓冲器满 (接收到了两个数据), 接收移位寄存器又有数据, 若此时检测到一个新的起始位, 则产生数据溢出事件, 此标志位一直有效直到接收缓冲器 (UDR) 被读取。对 UCSRA 进行写入操作时, 这一位要置“0”。
- **PE**: 奇偶校验错误标志位。当奇偶校验使能 ( $UPM1 = 1$ ), 且接收缓冲器中所接收到的下一个字符有奇偶校验错误时, PE 被置位, 该标志位一直有效直到接收缓冲器 (UDR) 被读取。对 UCSRA 进行写入操作时, 这一位要置“0”。
- **U2X**: 倍速发送设置位。该标志位仅对异步操作有影响, 使用同步操作时需要将此位清零。此位置“1”时可将波特率分频因子从 16 降到 8, 从而有效地将异步通信模式的传输速率加倍。
- **MPCM**: 多处理器通信模式设置位。设置此位将启动多处理器通信模式, MPCM 被置“1”后, USART 接收器接收到的那些不包含地址信息的输入帧都将被忽略。发送器不受 MPCM 设置的影响, 详细内容请参考后续章节的“多处理器通信模式”。

### 7.2.3 串口控制和状态寄存器 B (UCSRB)

USART 控制和状态寄存器 B 主要用于对 USART 进行各种设置, 其内部结构如表 7.3 所示。

表 7.3 USART 控制和状态寄存器 B

BIT	7	6	6	4	3	2	1	0
位	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
初始值	0	0	0	0	0	0	0	0



- **RXCIE**: 接收结束中断使能控制位, 置“1”后使能 RXC 中断, 当 RXCIE 为“1”且全局中断标志位 SREG 置“1”、UCSRA 寄存器的 RXC 也为“1”时, 在 USART 接收结束时产生中断事件。
- **TXCIE**: 发送结束中断使能控制位。置“1”后使能 TXC 中断, 当 TXCIE 为“1”且全局中断标志位 SREG 置“1”、UCSRA 寄存器的 TXC 也为“1”时, 在 USART 发送结束时产生中断事件。
- **UDRIE**: ATmega16 串口数据寄存器空中断使能控制位。置“1”后使能 UDRE 中断, 当 UDRIE 为“1”、全局中断标志位 SREG 置“1”、UCSRA 寄存器的 UDRE 也为“1”时, 在 USART 数据寄存器为空时产生中断事件。
- **RXEN**: ATmega16 串口接收使能控制位。置“1”后将启动串口接收器, 外部 RXD 引脚的通用端口功能被 USART 功能所取代, 当禁止时, 接收器将刷新接收缓冲器, 并且使 FE、DOR 及 PE 标志位无效。
- **TXEN**: ATmega16 串口发送使能控制位。置“1”后将启动 USART 发送器, 外部 TXD 引脚的通用端口功能被 USART 功能所取代。如果 TXEN 被清零, 只有等到所有的数据发送完成后发送器才能够真正禁止, 即发送移位寄存器与发送缓冲寄存器中没有要传送的数据。发送器禁止后, 外部 TxD 引脚恢复其通用 I/O 功能。
- **UCSZ2**: 字符长度控制位。UCSZ2 与 UCSRC 寄存器的 UCSZ1、UCSZ0 位组合起来设置数据帧所包含的数据位数 (字符长度)。
- **RXB8**: 接收数据的第 8 位。当接收数据为 9 位时, RXB8 是第 9 个数据位, 读取 UDR 包含的低位数据之前, 应先读取 RXB8。
- **TXB8**: 发送数据的第 8 位。当发送数据为 9 位时, TXB8 是第 9 个数据位, 写 UDR 之前, 应先将第 9 位数据写入该位。

#### 7.2.4 串口控制和状态寄存器 C (UCSRC)

USART 控制和状态寄存器 C 也是用于对 USART 进行相关设置的, 其内部结构如表 7.4 所示。

表 7.4 USART 控制和状态寄存器 C

BIT	7	6	6	4	3	2	1	0
位	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	1	0	0	0	0	1	1	0



#### 注意

UCSRC 寄存器与 UBRRH 寄存器共用相同的寄存器地址。

- **URSEL**: 寄存器选择控制位。对该位的设置可以选择访问 UCSRC 寄存器或 UBRRH 寄存器, 当该位被设置为“1”时, 选择访问 UCSRC 寄存器。





- **UMSEL**: ATmega16 串口模式选择控制位, 用于设置同步或异步工作模式。当此位为“0”时, 选择异步工作模式; 当此位为“1”时, 选择同步工作模式。
- **UPM1 和 UPM0**: ATmega16 串口的奇偶校验模式控制位, 用于设置奇偶校验的工作模式并使能奇偶校验。如果使能奇偶校验, 在发送数据时, 发送器都会自动产生并发送奇偶校验位。对每一个接收到的数据, 接收器都会产生一个奇偶校验值, 并与 UPM0 所设置的值进行比较。如果不匹配则将 UCSRA 中的 PE 置位, 当 UPM1 ~ UPM10 被设置为“00”时, 禁止奇偶校验; 当设置为“10”时, 进行偶校验; 当设置为“11”时, 进行奇校验; “01”为保留设置, 不使用。
- **USBS**: 停止位。当 ATmega16 工作为接收模式时, 忽略这一位的设置; 当此位被置“1”时, 选择 2 位停止位, 清“0”时为 1 位停止位。
- **UCSZ1/0**: 字符长度控制位, 与 UCSRB 寄存器的 UCSZ2 结合在一起可以设置数据帧包含的数据位数 (字符长度), 如表 7.5 所示。

表 7.5 USART 的字符长度控制

UCSZ2	UCSZ1	UCSZ0	字符长度
0	0	0	5 位
0	0	1	6 位
0	1	0	7 位
0	1	1	8 位
1	0	0	保留
1	0	1	保留
1	1	0	保留
1	1	1	9 位

- **UCPOL**: 时钟极性控制位, 仅在同步工作模式下有效。当使用异步模式时, 将这一位清零。UCPOL 设置了输出数据的改变和输入数据采样以及同步时钟 XCK 之间的关系。当 UCPOL 被置“1”时, 外部 TXD 引脚上的发送数据在 XCK 的上升沿有效, 外部 RXD 引脚的输入数据在 XCK 的下降沿有效; 当 UCPOL 被清零时, 外部 TXD 引脚上的发送数据在 XCK 的下降沿有效, 外部 RXD 引脚的输入数据在 XCK 的上升沿有效。

### 7.2.5 串口波特率寄存器 (UBRRL 和 UBRRH)

ATmega16 串口波特率寄存器用设置串口的通信波特率, 其内部结构如表 7.6 所示。

表 7.6 USART 的波特率寄存器

URSEL	—	—	—	USRR[11~8]			
USRR[7~0]							
R/W	R	R	R	R/W	R/W	R/W	R/W

续表

URSEL	—	—	—	USRR[ 11 ~ 8 ]			
USRR[ 7 ~ 0 ]							
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0



注意

UCSRC 寄存器与 UBRRH 寄存器共用相同的寄存器地址。

- URSEL：寄存器选择位，通过该位选择访问 UCSRC 寄存器或 UBRRH 寄存器，当操作 UBRRH 寄存器时，该位清零。
- USRRH14 ~ USRRH12：保留位，是为以后的使用而保留的，为了与以后的器件兼容，写 UBRRH 寄存器时将这些位清零。
- UBRH11 ~ UBRH0：ATmega16 串口波特率寄存器，该 12 位的寄存器用于设置 USART 的波特率信息，其中 UBRRH 为波特率设置高 4 位，UBRRL 为低 8 位。波特率的改变将造成正在进行的数据传输受到破坏，对 UBRRL 寄存器的写操作将立即更新波特率分频器。

表 7.7 至表 7.10 为常用频率的晶振以及谐振器异步工作模式下的 UBRR 寄存器设置参数，其中，误差可以通过如下公式来计算：

$$\text{Error}[\%] = \left( \frac{\text{BaudRate}_{\text{ClosestMatch}}}{\text{BaudRate}} - 1 \right) \times 100\%$$

表 7.7 ATmega16 的常用波特率设置参数 (1)

波特率 (b/s)	$F_{\text{osc}} = 1.0000\text{MHz}$				$F_{\text{osc}} = 1.8432\text{MHz}$				$F_{\text{osc}} = 2.0000\text{MHz}$			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	BRR	误差	BRR	误差	BRR	误差	BRR	误差	BRR	误差	BRR	误差
2400	26	0.2%	61	0.2%	47	0.0%	96	0.0%	61	0.2%	103	0.2%
4800	12	0.2%	26	0.2%	23	0.0%	47	0.0%	26	0.2%	61	0.2%
9600	6	-7.0%	12	0.2%	11	0.0%	23	0.0%	12	0.2%	26	0.2%
14.4k	3	8.6%	8	-3.6%	7	0.0%	16	0.0%	8	-3.6%	16	2.1%
19.2k	2	8.6%	6	-7.0%	6	0.0%	11	0.0%	6	-7.0%	12	0.2%
28.8k	1	8.6%	3	8.6%	3	0.0%	7	0.0%	3	8.6%	8	-3.6%
38.4k	1	-18.6%	2	8.6%	2	0.0%	6	0.0%	2	8.6%	6	-7.0%
67.6k	0	8.6%	1	8.6%	1	0.0%	3	0.0%	1	8.6%	3	8.6%
76.8k	—	—	1	-18.6%	1	-26.0%	2	0.0%	1	-18.6%	2	8.6%
116.2k	—	—	0	8.6%	0	0.0%	1	0.0%	0	8.6%	1	8.6%
230.4k	—	—	—	—	—	—	—	—	—	—	—	—
260k	—	—	—	—	—	—	—	—	—	—	0	0.0%
最大	62.6kb/s		126kb/s		116.2kb/s		230.4kb/s		126kb/s		260kb/s	



表 7.8 ATmega16 的常用波特率设置参数 (2)

波特率 (b/s)	$F_{\text{ocs}} = 3.6864\text{MHz}$				$F_{\text{ocs}} = 4.0000\text{MHz}$				$F_{\text{ocs}} = 7.3728\text{MHz}$			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	BRR	误差	BRR	误差	UBRR	误差	UBRR	误差	BRR	误差	BRR	误差
2400	96	0.0%	191	0.0%	103	0.2%	207	0.2%	191	0.0%	383	0.0%
4800	47	0.0%	96	0.0%	61	0.2%	103	0.2%	96	0.0%	191	0.0%
9600	23	0.0%	47	0.0%	26	0.2%	61	0.2%	47	0.0%	96	0.0%
14.4k	16	0.0%	31	0.0%	16	2.1%	34	-0.8%	31	0.0%	63	0.0%
19.2k	11	0.0%	23	0.0%	12	0.2%	26	0.2%	23	0.0%	47	0.0%
28.8k	7	0.0%	16	0.0%	8	-3.6%	16	2.1%	16	0.0%	31	0.0%
38.4k	6	0.0%	11	0.0%	6	-7.0%	12	0.2%	11	0.0%	23	0.0%
67.6k	3	0.0%	7	0.0%	3	8.6%	8	-3.6%	7	0.0%	16	0.0%
76.8k	2	0.0%	6	0.0%	2	8.6%	6	-7.0%	6	0.0%	11	0.0%
116.2k	1	0.0%	3	0.0%	1	8.6%	3	8.6%	3	0.0%	7	0.0%
230.4k	0	0.0%	1	0.0%	0	8.6%	1	8.6%	1	0.0%	3	0.0%
260k	0	-7.8%	1	0.0%	0	0.0%	1	0.0%	1	-7.8%	3	-7.8%
0.6M	—	—	0	-7.8%	—	—	0	0.0%	0	-7.8%	1	-7.8%
1M	—	—	—	-7.8%	—	—	—	—	—	—	0	-7.8%
最大	230.4kb/s		460.8kb/s		260kb/s		0.6Mb/s		460.8kb/s		921.6kb/s	

表 7.9 ATmega16 的常用波特率设置参数 (3)

波特率 (b/s)	$F_{\text{ocs}} = 8.0000\text{MHz}$				$F_{\text{ocs}} = 11.0692\text{MHz}$				$F_{\text{ocs}} = 14.7466\text{MHz}$			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	BRR	误差	BRR	误差	BRR	误差	BRR	误差	BRR	误差	BRR	误差
2400	207	0.2%	416	-0.1%	287	0.0%	676	0.0%	383	0.0%	767	0.0%
4800	103	0.2%	207	0.2%	143	0.0%	287	0.0%	191	0.0%	383	0.0%
9600	61	0.2%	103	0.2%	71	0.0%	143	0.0%	96	0.0%	191	0.0%
14.4k	34	-0.8%	68	0.6%	47	0.0%	96	0.0%	63	0.0%	127	0.0%
19.2k	26	0.2%	61	0.2%	36	0.0%	71	0.0%	47	0.0%	96	0.0%
28.8k	16	2.1%	34	-0.8%	23	0.0%	47	0.0%	31	0.0%	64	0.0%
38.4k	12	0.2%	26	0.2%	17	0.0%	36	0.0%	23	0.0%	47	0.0%
67.6k	8	-3.6%	16	2.1%	11	0.0%	23	0.0%	16	0.0%	31	0.0%
76.8k	6	-7.0%	12	0.2%	8	0.0%	17	0.0%	11	0.0%	23	0.0%
116.2k	3	8.6%	8	-3.6%	6	0.0%	11	0.0%	7	0.0%	16	0.0%
230.4k	1	8.6%	3	8.6%	2	0.0%	6	0.0%	3	0.0%	7	0.0%
260k	1	0.0%	3	0.0%	2	-7.8%	6	-7.8%	3	-7.8%	6	-6.3%
0.6M	0	0.0%	1	0.0%	—	—	2	-7.8%	1	-7.8%	3	-7.8%
1M	—	—	0	0.0%	—	—	—	—	0	-7.8%	1	-7.8%
最大	0.6Mb/s		1Mb/s		691.2kb/s		1.3824Mb/s		921.6kb/s		1.8432Mb/s	



表 7.10 ATmega16 的常用波特率设置参数 (4)

波特率 (b/s)	$F_{\text{ocs}} = 16.0000\text{MHz}$				$F_{\text{ocs}} = 18.4320\text{MHz}$				$F_{\text{ocs}} = 20.0000\text{MHz}$			
	U2X = 0		U2X = 1		U2X = 0		U2X = 1		U2X = 0		U2X = 1	
	UBRR	误差	UBRR	误差	UBRR	误差	UBRR	误差	UBRR	误差	UBRR	误差
2400	416	-0.1%	832	0.0%	479	0.0%	969	0.0%	620	0.0%	1041	0.0%
4800	207	0.2%	416	-0.1%	239	0.0%	479	0.0%	269	0.2%	620	0.0%
9600	103	0.2%	207	0.2%	119	0.0%	239	0.0%	129	0.2%	269	0.0%
14.4k	68	0.6%	138	-0.1%	79	0.0%	169	0.0%	86	-0.2%	173	0.2%
19.2k	61	0.2%	103	0.2%	69	0.0%	119	0.0%	64	0.2%	129	-0.2%
28.8k	34	-0.8%	68	0.6%	39	0.0%	79	0.0%	42	0.9%	86	0.2%
38.4k	26	0.2%	61	0.2%	29	0.0%	69	0.0%	32	-1.4%	64	-0.2%
67.6k	16	2.1%	34	-0.8%	19	0.0%	39	0.0%	21	-1.4%	42	0.2%
76.8k	12	0.2%	26	0.2%	14	0.0%	29	0.0%	16	1.7%	32	0.9%
116.2k	8	-3.6%	16	2.1%	9	0.0%	19	0.0%	10	-1.4%	21	-1.4%
230.4k	3	8.6%	8	-3.6%	4	0.0%	9	0.0%	4	8.6%	10	-1.4%
260k	13	0.0%	7	0.0%	4	-7.8%	8	2.4%	4	0.0%	9	-1.4%
0.6M	1	0.0%	3	0.0%	—	—	4	-7.8%	—	—	4	0.0%
1M	0	0.0%	1	0.0%	—	—	—	—	—	—	—	—
最大	1Mb/s		2Mb/s		1.162kb/s		2.304Mb/s		1.26Mb/s		2.6Mb/s	

## 7.3 ATmega16 串口的使用方法

ATmega16 串口的使用包括选择时钟源、选择数据帧格式、发送和接收数据以及多机通信几部分。

### 7.3.1 选择 ATmega16 串口的时钟源

ATmega16 串口的时钟发生器为发送和接收提供基础时钟，其支持 4 种模式的时钟：正常的异步模式、倍速的异步模式、主机同步模式和从机同步模式。

ATmega16 串口的控制和状态寄存器 C (UCSRC) 中的 UMSEL 位用于选择串口的异步工作模式或者同步工作模式；而控制和状态寄存器 A (UCSRA) 中的 U2X 位用于选择串口的倍速工作模式和普通工作模式，需要注意的是倍速工作模式只能在异步工作模式下才有效，当串口处于同步工作模式 (UMSEL = 1) 时，XCK 的数据方向寄存器 (DDR\_XCK) 决定了其时钟源是由内部产生（主机模式）还是由外部输入（从机模式）的，仅在同步工作模式下 XCK 有效。

#### 1. 片内时钟源

ATmega16 的片内时钟源可以用于异步工作模式以及同步工作主机模式，在这种工作模式下串口波特率寄存器 UBRR 和降序计数器相连接，一起构成可编程的预分频器或波特率

发生器。降序计数器对系统时钟计数，当其计数到 0 或 UBRRL 寄存器进行写操作时，会自动装入 UBRR 寄存器的值。当计数到零时，产生一个时钟作为波特率发生器的输出时钟，这个输出时钟的频率为  $F_{\text{osc}} / (\text{UBRR} + 1)$ 。串口的发送模块对波特率发生器的输出时钟进行 2、8 或 16 的分频，具体情况取决于具体的工作模式；波特率发生器的输出被直接用于串口的接收器与数据恢复单元。数据恢复单元使用了一个有 2、8 或 16 个状态的状态机，具体状态数由 UMSEL、U2X 与 DDR\_XCK 位设定的工作模式决定。表 7.11 为 ATmega16 的串口波特率计算公式列表。

表 7.11 ATmega16 的波特率计算公式

工作模式	波特率计算公式	UBRR 初始化值计算公式
异步工作正常倍速 (U2X = 0)	$\text{BAUD} = \frac{F_{\text{osc}}}{16 \times (\text{UBRR} + 1)}$	$\text{UBRR} = \frac{F_{\text{osc}}}{16 \times \text{BAUD}} - 1$
异步工作倍速 (U2X = 1)	$\text{BAUD} = \frac{F_{\text{osc}}}{8 \times (\text{UBRR} + 1)}$	$\text{UBRR} = \frac{F_{\text{osc}}}{8 \times \text{BAUD}} - 1$
同步主机	$\text{BAUD} = \frac{F_{\text{osc}}}{2 \times (\text{UBRR} + 1)}$	$\text{UBRR} = \frac{F_{\text{osc}}}{2 \times \text{BAUD}} - 1$



## 说明

其中， $F_{\text{osc}}$  为系统工作时钟，BAUD 的单位为 b/s（每秒位传播速度）。

通过将 UCSRA 寄存器中的 U2X 的值置“1”，可以使得串口的传播速度提高一倍，这种工作方式称为倍速模式，但是只能在异步工作模式下使用，在同步工作模式下该位必须为“0”。通过设置 U2X，可以把波特率分频器的分频值从 16 降低到 8，从而使得异步通信的传输速率加倍，此时 USART 的接收器只使用一半的采样数对数据进行采样及时钟恢复，因此在该模式下需要更精确的系统时钟与更精确的波特率设置，对于 USART 的发送器则没有这个要求。

## 2. 片外时钟源

在外部时钟同步从机操作模式下，ATmega16 的串口由输入到 XCK 引脚的片外时钟驱动，输入到 XCK 引脚的外部时钟由同步寄存器进行采样，用以提高稳定性。同步寄存器的输出通过一个边沿检测器，然后同时应用于 USART 的发送器与接收器，由于这一过程引入了两个时钟周期的延时，因此外部 XCK 输入的最大时钟频率不能超过 ATmega16 工作频率的 1/4，受到了一定的限制：

$$F_{\text{XCK}} < \frac{F_{\text{osc}}}{4}$$



## 注意

$F_{\text{osc}}$  由系统时钟的稳定性决定，为了防止因频率漂移而丢失数据，建议保留足够的裕量。

## 3. 同步工作模式下的时钟

当 ATmega16 的串口位于同步工作模式 (UMSEL = 1) 时，时钟将从 XCK 引脚输入（从机



模式）或者输出（主机模式），在改变数据输出端 TXD 的 XCK 时钟的相反边沿，ATmega16 对数据输入端 RXD 进行采样，其过程如图 7.2 所示。

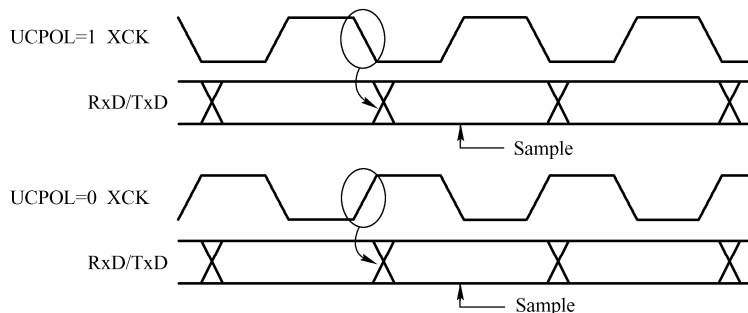


图 7.2 同步工作模式下的时钟时序

UCRSC 寄存器的 UCPOL 位用于修改使用 XCK 时钟的哪个边沿对数据进行采样和改变输出数据。当 UCPOL=0 时，在 XCK 的上升沿改变输出数据，在 XCK 的下降沿进行数据采样；当 UCPOL=1 时，在 XCK 的下降沿改变输出数据，在 XCK 的上升沿进行数据采样。

### 7.3.2 选择 ATmega16 串口的数据帧格式

ATmega16 的串口数据帧由数据、同步位（起始位与停止位）以及用于纠错的奇偶校验位组成，它接受以下几种组合的数据帧格式，以起始位开始，紧接着是最多为 9 个数据位的数据字最低位，以停止位结束。如果使能了校验位，那么校验位将紧接着数据位，位于数据位和停止位之间。当一个完整的数据帧传输后，可以立即传输下一个新的数据帧，或使传输线处于空闲状态。

- 1 个起始位；
- 5、6、7、8 或 9 个数据位；
- 无校验位、奇校验或偶校验位；
- 1 或 2 个停止位。

USART 的串行数据帧组成如图 7.3 所示。

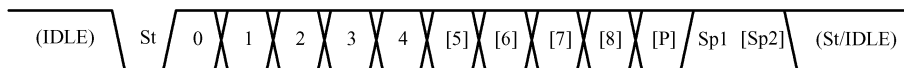


图 7.3 USART 的串行数据帧组成

- St: 起始位，1 位，总是为低电平，必须。
- N: 数据位 0 至数据位 8，其中 5 ~ 8 位非必须。
- P: 校验位，1 位，可以为奇校验或偶校验，非必须。
- Sp: 停止位，1 位或者 2 位，总是为高电平，第 1 位必须，第 2 位非必须。
- IDLE: 表示通信线上没有数据传输，线路空闲时必须为高电平。

ATmega16 的串口数据帧的结构由 UCSRB 和 UCSRC 寄存器中的 UCSZ2 ~ UCSZ0、UPM1 ~ UPM0、USBS 位决定，其接收器与发送器使用相同的设置。对这些位的任何改变都





可能破坏正在进行的数据传送与接收。其中，USART 的字长度位 UCSZ2 ~ UCSZ0 确定了数据帧的数据位长度；校验模式位 UPM1 ~ UPM0 决定了是否使能校验位以及决定校验的类型；USBS 位用于设置帧的结束位位数，这些位的具体用法可以参考 6.1.1 节。



#### 注意

USART 的接收器忽略第 2 个停止位，因此帧错误（FE）只在第 1 个结束位为“0”时有效。

ATmega16 串口的校验位的计算方式是对有效数据的各个位进行异或运算，如果选择了奇校验，则该异或结果还需要取反，其公式如下：

$$P_{\text{even}} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$

$$P_{\text{odd}} = d_{n-1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

式中： $P_{\text{even}}$ ：偶校验结果； $P_{\text{odd}}$ ：奇校验结果； $d_n$ ：第  $n$  个数据。

### 7.3.3 ATmega16 串口的数据收发

ATmega16 串口数据收发包括初始化、数据发送和数据接收三部分。

#### 1. ATmega16 串口的初始化

ATmega16 串口初始化过程通常包括波特率的设定、帧结构的设定，使能接收器或发送器以及根据需要对相关的中断寄存器进行设置和清除。

ATmega16 串口的初始化设置应该在没有数据传输的情况下进行，因为对相关寄存器的操作会导致数据的传输异常，通常可以使用 TXC 标志位来判断一个数据帧的发送是否完成，使用 RXC 标志位来检验接收缓冲器中是否还有数据未读出，所以在每次将数据写入数据寄存器 UDR 之前，必须将 TXC 标志位清零。

#### 2. ATmega16 串口的数据发送

可以通过对 UCSRB 寄存器的发送允许位 TXEN 置“1”来使能 ATmega16 的串口数据发送，此后 TXD 引脚成为串行数据发送引脚，它通用 I/O 功能被串口发送功能所取代，UDR 寄存器中的数据通过该引脚串行送出。

在发送数据之前要设置好波特率、工作模式与帧结构；如果使用同步发送模式，施加于 XCK 引脚上的时钟信号即为数据发送的时钟。

如果待发送的数据帧为 6 ~ 8 位，将需要发送的数据写入 UDR 将启动数据发送，当移位寄存器可以发送新一帧数据时，缓冲的数据将转移到移位寄存器；当移位寄存器处于空闲状态（没有正在进行的数据传输），或前一帧数据的最后一个停止位传送结束，将加载新的数据，一旦移位寄存器加载了新的数据，就会按照设定的波特率完成数据的发送。

如果需要发送包含 9 位数据的数据帧，则应该设置“UCSZ2 ~ UCSZ0 = 111”，并且应该先将数据的第 9 位写入寄存器 UCSRB 的 TXB8 位置，然后再将低 8 位数据写入发送数据寄存器 UDR。

ATmega16 的串口发送器有数据寄存器空标志（UDRE）以及传输结束标志（TXC）两个标志位，它们都可以用于产生中断事件。

数据寄存器空（UDRE）标志位用于表示发送缓冲器是否可以接受一个新的数据，该位在发送缓冲器为空，即没有数据等待发送时被置“1”；当发送缓冲器存在需要发送的数据时清零，当对 UCSRA 寄存器进行写操作时，该位要写“0”。当 UCSRB 寄存器中的数据寄存器空中断使能位 UDRIE 被置“1”，且全局中断使能时，如果 UDRE 被置位，将产生 USART 数据寄存器空中断事件。对寄存器 UDR 的写操作将清零 UDRE 位，当采用中断方式的传输数据时，在数据寄存器空中断服务程序中必须写一个新的数据到 UDR 寄存器，以清零 UDRE 位，或者禁止数据寄存器空中断，否则一旦该中断程序结束，一个新的中断将立即再次产生。

当整个发送数据帧移出发送移位寄存器，同时发送缓冲器中又没有新的数据时，发送结束标志 TXC 置“1”，TXC 位在传送结束中断执行时自动清零，也可通过向该位写入“1”来清除这个标志位。当 UCSRB 上的发送结束中断使能位 TXCIE 与全局中断使能位均被置“1”时，如果 TXC 标志位的置位则会产生一个 USART 发送结束中断事件。在进入中断服务程序后，TXC 标志位被自动清零，中断处理程序不需要执行 TXC 清零操作。

ATmega16 的内部奇偶校验产生电路为串行数据帧生成相应的校验位，当校验位被使能（UPM1 = 1）时，发送控制逻辑电路会在数据的最后一位与第一个停止位之间插入奇偶校验位。

当 TXEN 位被清零后，只有等到所有的数据发送完成后，发送器才能够真正禁止，即发送移位寄存器与发送缓冲寄存器中没有要传送的数据。

在关闭 ATmega16 串口的发送器后，TXD 引脚恢复其通用 I/O 功能。

### 3. ATmega16 串口的数据接收

可以通过对 UCSRB 寄存器的接收允许位 RXEN 置“1”来启动 ATmega16 串口接收器，当使能后，外部引脚 RXD 成为串行数据输入引脚，其通用 I/O 功能被串口发送功能所取代，数据通过该引脚进入接收缓冲器。

进行数据接收之前首先要设置好波特率、工作模式及帧格式，如果使用同步操作，XCK 引脚上的时钟信号即为数据接收的时钟。

如果待接收的数据帧为 6 到 8 个数据位，当接收器检测到一个有效的起始位时，开始接收数据。起始位后的每一位数据都将以所设定的波特率或 XCK 时钟脉冲进行接收，直到收到一帧数据的第一个停止位为止。接收到的数据被送入接收移位寄存器，第二个停止位会被接收器忽略。当接收到第一个停止位后，接收移位寄存器中就包含了一个完整的数据帧。这时移位寄存器中的内容将被转移到接收缓冲器中，通过读取 UDR 寄存器就可以获得接收缓冲器的内容。

如果需要接收 9 位数据的数据帧，则需要设置“UCSZ2 ~ UCSZ0 = 111”，而且从 UDR 读取低 8 位之前必须首先读取寄存器 UCSRB 的 RXB8 位以获得第 9 位数据，这个规则同样适用于状态标志位 FE、DOR 及 UPE。ATmega16 串口的状态通过读取 UCSRA 寄存器获得，而数据通过读取 UDR 寄存器获得。

接收结束标志位（RXC）用来标志 ATmega16 的串口接收缓冲器中是否有未读出的数据，当接收缓冲器中有未读出的数据时，此位为“1”；当接收缓冲器空没有未读出的数据时，此位为“0”。

如果接收器被禁止  $RXEN = 0$ ，则接收缓冲器会被刷新，从而使  $RXC$  位清零。

置位  $UCSRB$  的接收结束中断使能位  $RXCIE$  后，只要  $RXC$  位被置“1”且全局中断使能，则会产生 ATmega16 串口接收结束中断事件，在使用中断方式进行数据接收时，数据接收结束中断服务程序必须从  $UDR$  读取数据以清  $RXC$  标志，否则只要中断处理程序一结束，一个新的中断就会产生。

ATmega16 的串口接收器提供了三个错误标志位置：帧错误标志（FE）、数据溢出标志（DOR）以及奇偶校验错误标志（UPE）。这三个标志位都位于寄存器  $UCSRA$  中，错误标志位与数据帧一起保存在接收缓冲器中，由于读取  $UDR$  会刷新接收缓冲器， $UCSRA$  的内容必须在接收缓冲器（ $UDR$ ）之前读入，所有的错误标志都不能产生中断。

- 帧错误标志（FE）用于表示存储在接收缓冲器中的下一个可读帧的第一个停止位的状态。如果停止位正确（为“1”）则 FE 标志为“0”，否则 FE 标志为“1”，这个标志可用来检测同步丢失、传输中断，也可用于协议处理。 $UCSRC$  中  $USBS$  位的设置不影响 FE 标志位，因为除了第一位，接收器忽略所有其他的停止位。
- 数据溢出标志（DOR）用于表示由于接收缓冲器满导致了数据丢失。当接收缓冲器满（包含了两个数据），接收移位寄存器又有数据，若此时检测到一个新的起始位，则产生数据溢出。DOR 标志位被置“1”表明在最近一次读取  $UDR$  寄存器和下一次读取  $UDR$  寄存器之间丢失了一个或更多的数据帧。当数据帧成功地从移位寄存器转入接收缓冲器后，DOR 标志被清零。
- 奇偶校验错误标志（UPE）用于表示接收缓冲器中的下一帧数据在接收时有奇偶错误，如果不使用奇偶校验，那么 UPE 位应该被清零，奇偶校验的计算方法参考 6.1.2 节以及本节。



#### 注意

在对寄存器  $UCSRA$  进行写操作时，这三位所对应的写入数据都必须为“0”。

奇偶校验工作方式位  $UPM1$  被置“1”时，奇偶校验器使能，而奇偶校验的模式由  $UPM0$  位确定。当奇偶校验被使能后，校验器将计算接收数据的奇偶并把结果与数据帧的奇偶位进行比较，校验结果将与数据、停止位一起存储在接收缓冲器中，此时就可以通过读取奇偶校验错误标志位（UPE）来判断接收到的帧中是否有奇偶错误。

如果下一个从接收缓冲器中读出的数据有奇偶错误，并且奇偶校验被使能（ $UPM1 = 1$ ），则 UPE 标志位被置位，直到接收缓冲器（ $UDR$ ）被读取之前，这一位将一直有效。

如果将接收使能位  $RXEN$  清零，ATmega16 串口的数据接收将被立刻禁止，正在接收的数据将丢失，接收器将不再占用  $RXD$  引脚，接收缓冲器 FIFO 也会被刷新，缓冲器中的数据将丢失。如果由于出错而必须在正常操作下刷新缓冲器，则需要一直读取  $UDR$  寄存器，直到  $RXC$  标志位被清零。

### 7.3.4 ATmega16 串口的多机通信

如果在一个系统中有多 ATmega16 需要通过串口模块进行数据通信时，置位  $UCSRA$  寄存器的多处理器通信模式位（MPCM）则可以对串口模块接收器接收到的数据帧进行过



滤，那些没有地址信息的帧将被忽略，也不会存入接收缓冲器，这种过滤有效地减少了需要 ATmega16 处理的数据帧的数量。

MPCM 位的设置不影响发送器的工作，但在使用多个 ATmega16 进行通信模式的系统中，它的使用方法会有所不同。

如果 ATmega16 串口接收器所接收的数据帧长度为 6 到 8 位，那么第 1 个停止位表示这一帧包含的是数据还是地址信息，如果接收器所接收的数据帧长度为 9 位，则由第 9 位（RXB8）来决定这个数据帧是数据信息还是地址信息。在这种情况下，如果用于确定帧类型的数据位（第 1 个停止位或第 9 个数据位）为“1”，则该数据帧为地址帧，否则为数据帧。

在多处理器通信模式下，多个从处理器可以从一个主处理器接收数据。首先要通过解码地址帧来确定所寻址的是哪一个处理器。如果寻址到某一个处理器，它将正常接收后续的数据，而其他的从处理器会忽略这些帧直到接收到下一个地址帧。

对于作为主机的 ATmega16 来说，在 9 位数据帧格式（UCSZ2 ~ UCSZ0 = 7）下，如果传输的是一个地址帧（TXB8 = 1），就将数据帧中的第 9 位（TXB8）置“1”，如果是一个数据帧，则将该位清零。



#### 注意

作为从处理器的 ATmega16 必须和作为主程序的 ATmega16 使用相同的数据帧格式。

多个 ATmega16 通过串口模块进行数据通信的步骤如下。

- (1) 所有作为从处理器的 ATmega16 置位 UCSRA 寄存器的 MPCM 位。
- (2) 作为主机的 ATmega16 发送地址帧，所有作为从处理器的 ATmega16 都会接收并读取此帧，此时这些从处理器的 UCSRA 寄存器的 RXC 位正常置位。
- (3) 每一个从机都会读取 UDR 寄存器的内容用以判断主机是否和自己通信，如果是，则清除 UCSRA 的 MPCM 位，否则将等待下一个地址数据帧的到来，并保持 MPCM 位为“1”。
- (4) 和主机通信的从机将接收所有的数据帧，直到收到一个新的地址帧，而那些保持 MPCM 位为“1”的从机将忽略这些数据。
- (5) 和主机通信的从机接收到最后一个数据帧后置位 MPCM 位，并等待主机发送下一个地址帧，然后回到第（2）步。

## 7.4 ATmega16 串口的应用实例

### 7.4.1 ATmega16 串口数据发送

本应用是一个 ATmega16 使用串口循环发送数据 0x00 ~ 0xFF 的实例。

#### 1. 实例的设计思路

对 ATmega16 的串口进行初始化，然后在主循环中采用软件延时并将一个软件定时器累加后通过串口发送。





## 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 7.4 所示, ATmega16 的 RXD (PD0) 和 TXD (PD1) 引脚分别连接到一个 COMPIM 模块, 并且在 TXD 引脚上连接了一个虚拟终端, 实例涉及的 Proteus 器件如表 7.12 所示。

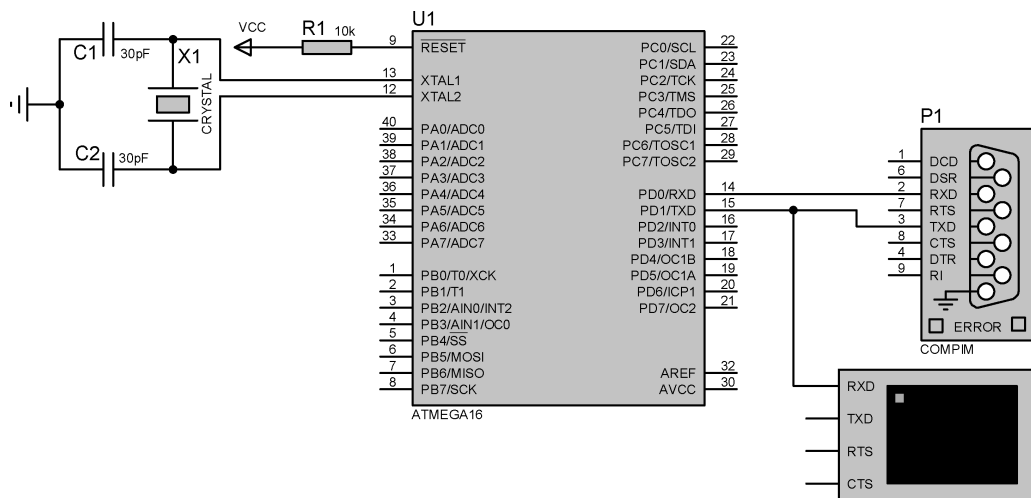


图 7.4 实例的 Proteus 电路

表 7.12 应用实例器件列表

器件名称	大类库	子类库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
COMPIM	Miscellaneous	—	串口模块

## 3. 实例的应用代码

实例的应用代码如例 7.1 所示, 由于 6.13 版本之后的 ICC AVR 的 STUDIO 库函数中没有提供相应的 putchar 函数, 所以代码自行构造了一个 putchar 函数用于串口的数据发送操作, 然后在主循环中对一个软件计数器 counter 计数, 并且将其计数值送出。

### 【例 7.1】 串口数据发送

```
#include <iom16v.h>
```

```
#include <macros.h>
```

```
unsigned char counter = 0x00;
```

```
//计数器初始化
```

```
//putchar 串口发送函数
```

```

extern void putchar( char inputdata)
{
    while( ! ( UCSRA&(1 << UDRE) ) );
    UDR = inputdata;
}

//延时函数
void delay( void)
{
    unsigned int i;
    for( i = 1 ; i < 100 ; i ++ ) ;
}

//1ms 延时函数
void delay_1ms( void)
{
    unsigned int i;
    for( i = 1 ; i < ( unsigned int ) ( 8 * 143 - 2 ) ; i ++ ) ;
}

//ms 延时函数
void DelayMs( unsigned int time)
{
    unsigned int i = 0;
    while( i < time)
    {
        delay_1ms( ) ;
        i ++ ;
    }
}

//端口初始化
void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0x00;
    PORTB = 0x00;
    DDRB = 0x00;
    PORTC = 0x00;
    DDRC = 0x00;
    PORTD = 0x00;
    DDRD = 0x00;
}

//UART 初始化函数, 9600 波特率
void uart0_init( void)

```



```

{
    UCSRB = 0x00;
    UCSRA = 0x00;
    UCSRC = BIT(URSEL) | 0x06;
    UBRRL = 0x33;
    UBRRH = 0x00;
    UCSRB = 0x48;
}

//uart 发送完成中断函数
#pragma interrupt_handler uart0_tx_isr:iv_USART0_TXC
void uart0_tx_isr( void)
{
    DelayMs( 50 );           //延时 50ms
    counter ++ ;           //计数器 ++
    putchar( counter );     //将计数器的值送出
}

//初始化 ATmega16
void init_devices( void)
{
    CLI( );
    port_init( );
    uart0_init( );

    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x00;
    SEI( );
}

//主程序
void main( void)
{
    init_devices( );
    putchar( 0x01 );        //发送一个 0x01
    DelayMs( 50 );          //延时
    while( 1 )
    {
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，在弹出的虚拟串行终端对话框（如果已经关闭，则可以在 Debug 中再次打

开) 中选择十六进制显示 (Hex Display Mode), 如图 7.5 所示, 此时可以看到对应的串口输出数据, 如图 7.6 所示。

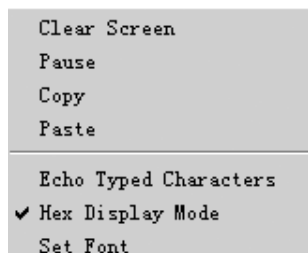


图 7.5 设置虚拟终端显示

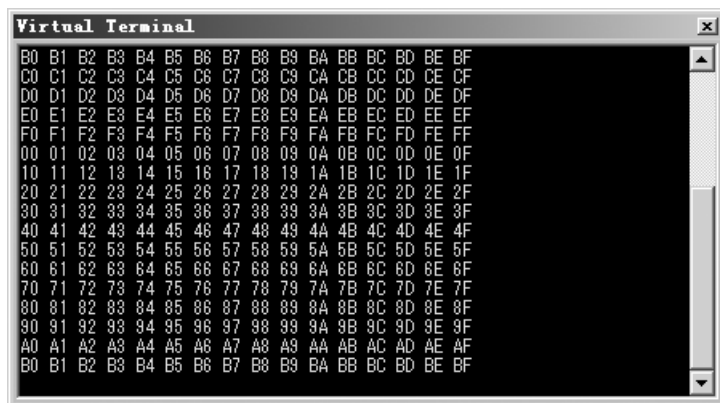


图 7.6 实例的仿真运行结果



## 总结

ICC AVR 的有关函数, 如 printf 等都要依赖于 putchar 函数, 所以读者可以自行将该函数添加进入函数库。

## 5. Proteus 中的 COMPIM 模块

COMPIM 模块是一个用于在 Proteus 中调试单片机 (不限于 ATmega16) 的模块, 其位于 Proteus 的 Miscellaneous 类库中, 如图 7.7 所示。

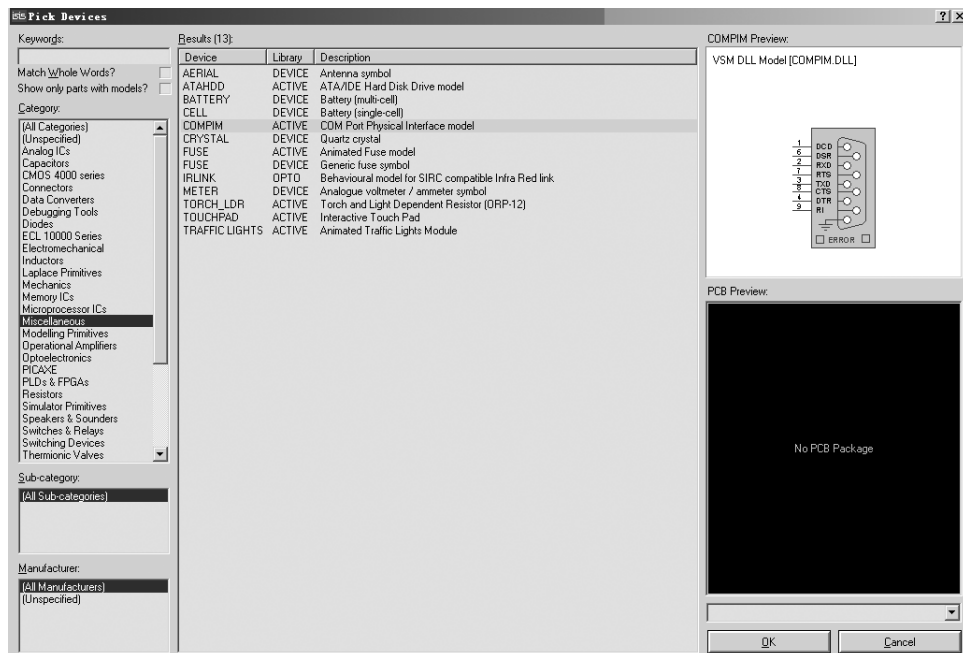


图 7.7 Proteus 中的 COMPIM 模块库



在使用 COMPIM 模块时,需要将其引脚和对应单片机引脚连接起来,通常来说,只需要连接 TXD (数据发送) 和 RXD (数据接收) 引脚即可,双击 COMPIM 模块会弹出如图 7.8 所示属性设置对话框,其中涉及的主要参数说明如下。

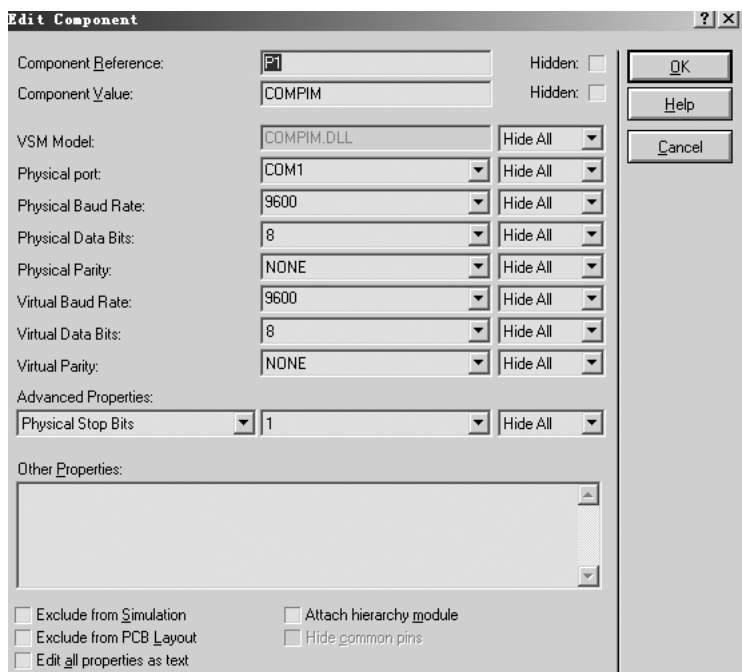


图 7.8 COMPIM 属性设置对话框

- Physical port: 物理端口,在和 PC 通信时,这个端口必须与 PC 的相应端口映射相同。
- Physical Baud Rate: 物理端口的波特率,可以选择为 50 ~ 57 600b/s。
- Physical Data Bits: 物理端口的数据位。
- Physical Parity: 物理端口的奇偶校验。
- Virtual Baud Rate: 虚拟串口的波特率。
- Virtual Data Bits: 虚拟串口的数据位。
- Virtual Parity: 虚拟端口的奇偶校验位。



#### 注意

通常来说,Physical 的参数和 Virtual 的参数设置为相同即可。

### 6. Proteus 中的虚拟终端

Proteus ISIS 提供了一个虚拟终端来进行串行通信的仿真,这相当于键盘和屏幕的双重功能,免去了上位机系统的仿真模型,使用户在用到单片机与上位机之间的串行通信时,直接由虚拟终端经 RS232 模型与单片机之间异步发送或接收数据。虚拟终端在运行仿真时会弹出一个仿真界面,当由 PC 向单片机发送数据时,可以和实际的键盘关联,用户可以从键盘经虚拟终端输入数据;当接收到单片机发送来的数据后,虚拟终端相当于一个显示屏,会

显示相应信息。

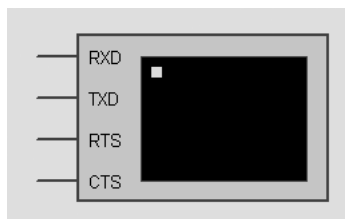


图 7.9 虚拟终端的模型

在 Proteus 中单击工具箱中的“Virtual Instrument Mode”按钮图标，在虚拟仪器对话框中选择虚拟终端“VIRTUAL TERMINAL”，在电路中单击，即可放置一个虚拟终端，其模型如图 7.9 所示，引脚说明如下。

- RXD：数据接收引脚。
- TXD：数据发送引脚。
- RTS：请求发送信号引脚。

- CTS：清除发送信号引脚，输出对 RTS 的响应信号。

在虚拟终端上双击，会弹出如图 7.10 所示的对话框，其主要参数说明如下。

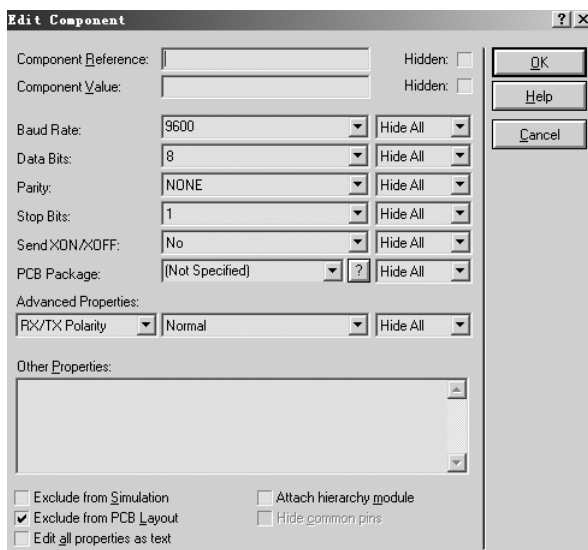


图 7.10 虚拟终端的属性设置对话框

- Baud Rate：波特率，范围为 300 ~ 57 600b/s。
- Data Bits：传输的数据位数，7 位或 8 位。
- Parity：奇偶校验位，包括奇校验、偶校验和无校验三种。
- Stop Bits：停止位，可以选择 1 或 2 位停止位。
- Send XON/XOFF：第 9 位发送允许/禁止控制。



#### 注意

虚拟终端不仅能显示串口发送的数据，还可以输入相应的数据通过串口进行发送，读者可以自行参考相应的资料。

### 7.4.2 和 PC 进行串行通信

本应用是一个使用 PC 的虚拟串口和 Proteus 仿真电路中的 ATmega16 串口进行数据通信

的实例。

### 1. 实例的设计思路

为了方便 Proteus 的串口调试, 可以使用虚拟串口软件 VSPD XP5 来为 PC 增加虚拟串口, VSPD XP5 是一个可以在 Windows 操作系统下为计算机系统增加一对在逻辑上“交叉相连的”虚拟串口的软件, 其使用步骤如下。

(1) 安装完成后启动 VSPD XP5, 如图 7.11 所示是其汉化版的启动界面。



图 7.11 启动 VSPD XP5

(2) 按照实际需求增加一个没有被占用的串口编号, 如图 7.12 所示, 并且点击“添加端口”按钮。



图 7.12 添加端口

(3) 此时在 VSPD XP5 的 Virtual ports 下就可以看到新增的两个虚拟串口, 如图 7.12 所

示的 COM3 和 COM4。需要注意的是，这两个虚拟串口是一对，在逻辑上是交叉连接的，也就是说 COM3 和 COM4 的 RXD 和 TXD 引脚是交叉对应连接在一起的，如果 COM3 发送一个字节的数据，COM4 将能收到。

(4) 双击打开 COMPIM 的设置对话框，将其映射到两个虚拟串口中的一个上，如图 7.13 所示映射到 COM3 上，并且设置好相关的通信参数，如波特率、数据位等。

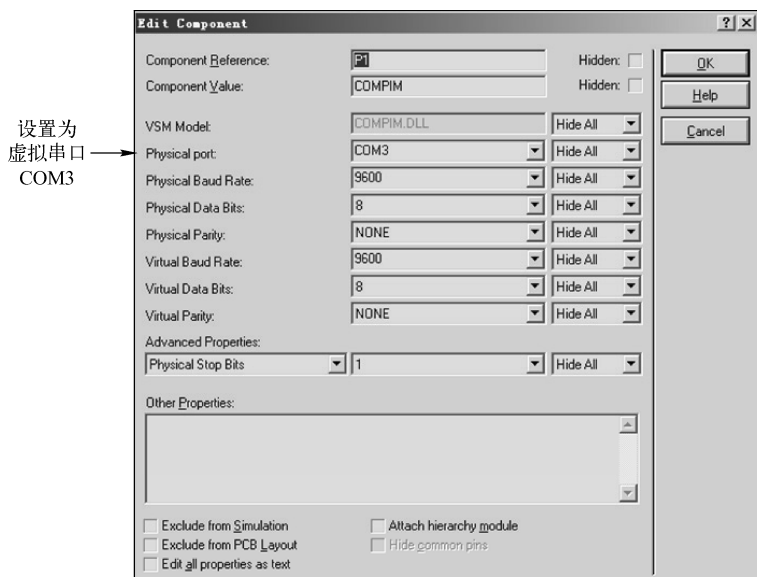


图 7.13 将 COMPIM 和虚拟串口映射

(5) 启动串口调试助手，将其映射到两个虚拟串口中的另外一个上，如图 7.14 所示映射为 COM4，并且参考串行端口 COMPIM 的属性设置好相应的通信参数。

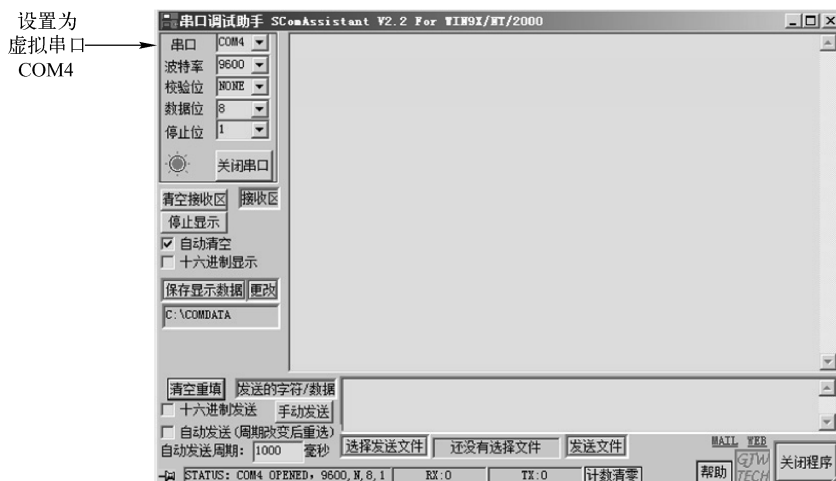


图 7.14 将串口调试助手和虚拟串口映射

**注意**

增加的虚拟串口编号必须不能和 PC 现有的串口冲突。

**2. 实例的 Proteus 电路图**

实例的 Proteus 电路如图 7.4 所示。

**3. 实例的应用代码**

实例的应用代码如例 7.2 所示。应用代码在中断服务子程序中接收 PC 通过串口发送的数据，然后调用 putchar 函数将其送回。

**【例 7.2】 和 PC 进行数据交互**

```
#include <iom16v.h>
#include <macros.h>
//putchar 串口发送函数
extern void putchar(char inputdata)
{
    while(!(UCSRA & (1 << UDRE)));
    UDR = inputdata;
}
//端口初始化函数
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
}
//串口初始化函数
void uart0_init(void)
{
    UCSRB = 0x00;
    UCSRA = 0x00;
    UCSRC = BIT(URSEL) | 0x06;
    UBRR1L = 0x33; //设置波特率为 9600b/s
    UBRRH = 0x00;
    UCSRB = 0x98;
```

```

}

//串口接收中断服务子函数
#pragma interrupt_handler uart0_rx_isr:iv_USART0_RXC
void uart0_rx_isr( void)
{
    unsigned char temp;
    temp = UDR;
    //读取串口接收到的数据
    putchar( temp );
    //调用 putchar 函数将数据送出
}

//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    uart0_init();
    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主程序
void main( void)
{
    init_devices();
    while(1)
    {
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，并且打开串口调试助手，发送一个字节的数据，可以同时虚拟中断和串口调试助手上看到相应数据反馈，如图 7.15 所示。



#### 总结

一定要注意 COMPIM、虚拟终端、VSPD XP5 和串口调试助手的相应串口号设置，VSPD XP5 虚拟出的两个串口可以看做是始终使用一根“数据线”交叉连接到一起的。

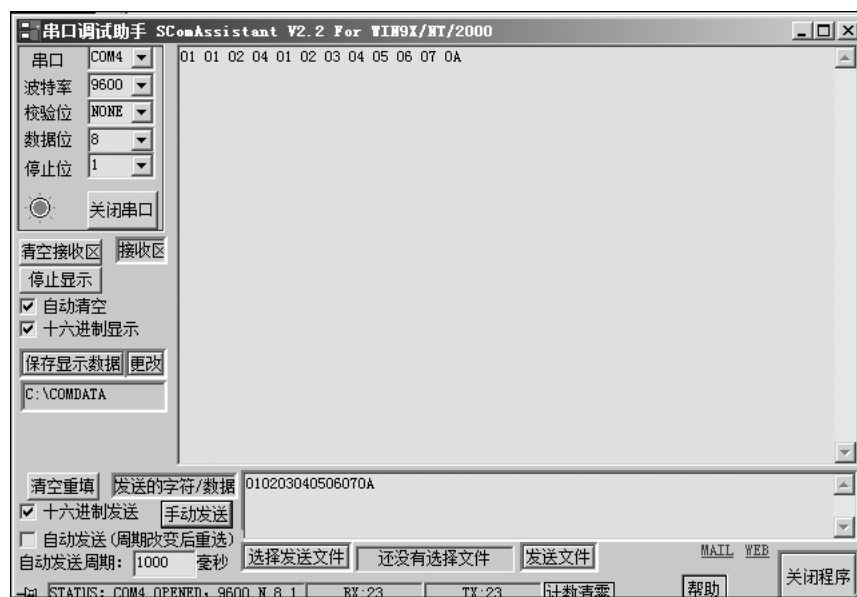


图 7.15 实例的仿真运行结果

## 第8章 ATmega16 单片机的 TWI 和 SPI 总线接口

ATmega16 单片机内部提供了一个 TWI 总线接口模块和一个 SPI 总线接口模块，用于和其他处理器以及外部器件进行通信。

### 8.1 TWI 总线基础

ATmega16 的 TWI 总线接口模块由内置的驱动电路和外部的 SCL 和 SDA 数据引脚构成，可以用于两个或者多个模块的高速串行数据交互。



注意

TWI（两线制接口）总线其实就是 I<sup>2</sup>C 总线，只是 ATMEL 公司为了规避专利权问题另外起了一个名字而已，其时序和控制方法和 I<sup>2</sup>C 总线完全相同。

TWI（I<sup>2</sup>C）接口总线有一些约定好的术语，用于描述在通信过程中所涉及的硬件和软件过程，这些术语如下。

- 主机：能产生 TWI 总线的 SCL 时钟信号，控制启动和停止传输的器件。
- 从机：和主机进行数据交换的器件，从机本身不能主动发起数据传输，也不能主动地产生时钟信号。
- 发送器：向 TWI 总线提供数据的器件。
- 接收器：从 TWI 总线上读取数据的器件。
- 多主机：同一条 TWI 总线上有一个以上可能会同时使用总线的主机。
- 主器件地址：主机的内部地址，每一种主器件有其特定且唯一的主器件地址。
- 从器件地址：从机的内部地址，每一种从器件有其特定且唯一的从器件地址。
- 仲裁过程：当同时有一个以上的主机尝试操作总线时，TWI 总线使得其中一个主机获得总线的使用权并不破坏报文的过程。
- 同步过程：两个或者两个以上器件同步时钟信号的过程。

#### 8.1.1 TWI 总线的数据交互过程

TWI 总线上的时钟信号 SCL 是由所有挂接到该信号线上器件的 SCL 信号进行逻辑“与”产生的，当这些器件中任何一个 SCL 引脚上的电平被拉低之后，SCL 信号线就将一直保持低电平，只有当所有器件的 SCL 引脚都恢复到高电平之后，SCL 总线才能恢复为高电平状态，所以这个时钟信号长度由维持低电平时间最长的器件来决定。在下一个时钟周期内，第





一个 SCL 引脚被拉低的器件又再次将 SCL 总线拉低，这样就形成了连续的 SCL 时钟信号。

TWI 总线上，必须以主器件发送启动信号来启动一次数据传送，以主器件发送停止信号来结束一次数据传送，从器件收到启动信号之后需要发送应答信号来通知主器件已经完成一次数据接收。

TWI 总线的启动信号是在读/写信号之前，当 SCL 处于高电平时，SDA 从高到低的一个跳变。

TWI 总线的停止信号是当 SCL 处于高电平时，SDA 从低到高的一个跳变，用于标志一种操作的结束，即将结束所有的相关的通信，如图 8.1 所示是启动信号和停止信号的时序。

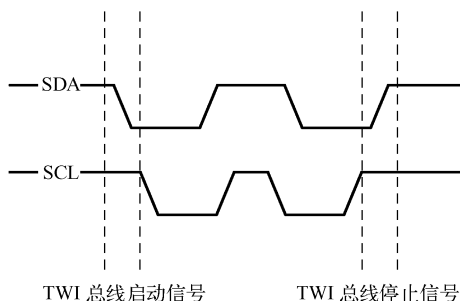


图 8.1 TWI 总线的启动信号和停止信号时序

TWI 总线上的主机在启动信号之后发送一个或者多个字节的数据，字节的高位在前，低位在后。主机在发送完成一个字节之后需要等待从机返回的应答信号。应答信号是从机在收到主机发送完的一个字节数据之后，在下次时钟到来时在 SDA 信号线上给出一个低电平，如图 8.2 所示。

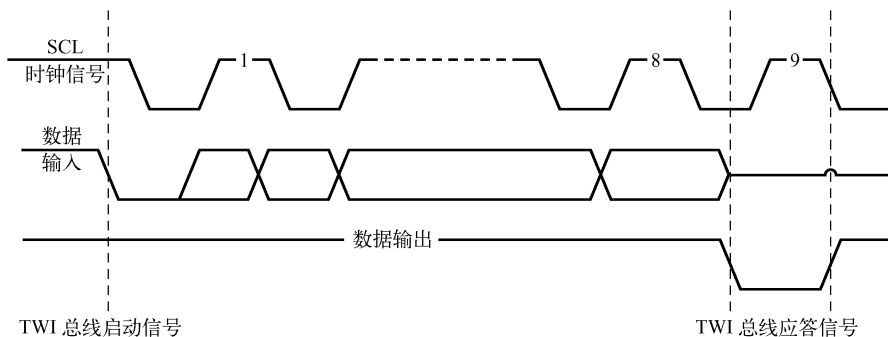


图 8.2 TWI 总线的应答信号时序

TWI 总线的数据通信过程必须按照如下流程进行。

(1) 主机发送启动信号，然后必须发送一个用于寻址的地址字节数据，包含需要和主机通信的从机地址。

(2) 当 SCL 时钟信号有效时，SDA 上的高电平代表该位数据为“1”，否则为“0”。

(3) 如果主机发送完一个字节数据之后还想继续发送数据，则可以不发送停止信号而是发送另一个启动信号，并且发送下一个地址字节以供连续通信，在连续通信完成之后发送



一个停止信号以结束通信。

TWI 总线的 SDA 和 SCL 信号线上均接有  $10\text{k}\Omega$  左右上拉电阻，当 SCL 为高电平时，对应的 SDA 的数据为有效；当 SCL 为低电平时，SDA 上的电平变化被忽略。当有启动信号之后，这条 TWI 总线被定义为“忙状态”，此时禁止同一条总线上其他没有获得总线控制权的主机操作该条总线，而在停止信号之后的一段时间内，总线被定义为“空闲状态”，此时允许其他主机通过总线仲裁来获得总线的使用权，进行下一次数据传送。

TWI 总线上可能会挂接几个都会对总线进行操作的主机，如果有一个以上的主机同时对总线进行操作时，总线就必须使用仲裁机制来决定哪一个主机能够获得总线的操作权。TWI 总线的仲裁是在 SCL 信号为高电平时，根据当前 SDA 状态来进行的。在总线仲裁期间，如果有其他的主机已经在 SDA 信号线上发送一个低电平，则发送高电平的主机将会发现该时刻 SDA 上的信号和自己发送的信号不一致，此时该主机则自动被仲裁为失去对总线的控制权，此过程如图 8.3 所示。

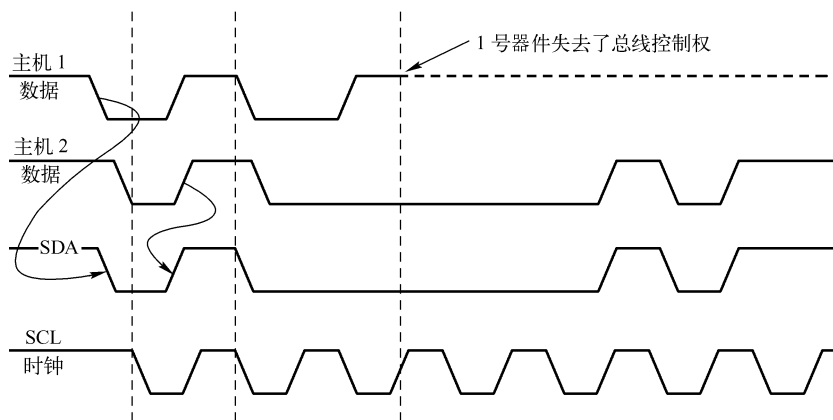


图 8.3 TWI 总线的仲裁过程

### 8.1.2 TWI 总线的地址

不同的 TWI 总线器件具有不同且唯一的地址，总线上的主机通过对这个地址的呼叫来确定对总线上的拥有该地址的器件进行数据交换，表 8.1 是 TWI 总线器件的地址分配示意，地址字节中前 7 位为该器件的 TWI 总线地址，该字节的第 8 位用来表明数据的传输方向，也称为读/写标志位。该标志位为“0”时为写操作，数据方向为主机到从机，读写位为“1”时为读操作，数据方向为从机到主机。

表 8.1 TWI 总线器件地址分配示意

地址最高位	地址第 6 位	地址第 5 位	地址第 4 位	地址第 3 位	地址第 2 位	地址第 1 位	R/W
-------	---------	---------	---------	---------	---------	---------	-----



#### 注意

TWI 总线协议中有广播地址，如果使用该地址进行数据通信，则在总线上的所有器件均能收到，具体信息可以参考 TWI 相关手册。



## 8.2 TWI 总线模块相关寄存器

ATmega16 的 TWI 总线模块由 SCL 和 SDA 引脚、比特率发生器单元、总线接口单元、地址匹配单元以及控制单元组成。ATmega16 使用相关的寄存器对 TWI 总线进行操作, 这些寄存器包括 TWI 比特率控制寄存器 (TWBR)、TWI 控制寄存器 (TWCR)、TWI 状态寄存器 (TWSR)、TWI 数据寄存器 (TWDR) 和 TWI 从机地址寄存器 (TWAR)。

### 8.2.1 比特率控制寄存器 (TWBR)

比特率控制寄存器 (TWBR) 用于存放 TWI 总线时钟 SCL 的分频因子, 用于在主机模式下产生相应的 SCL 信号, 其内部结构如表 8.2 所示。

表 8.2 比特率控制寄存器 TWBR

BIT	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

### 8.2.2 TWI 控制寄存器 (TWCR)

TWI 控制寄存器 (TWCR) 用来控制 TWI 的操作, 可以使能 TWI 模块、通过施加 START 到 TWI 总线上来启动主机访问、产生接收器应答、产生 STOP 状态, 以及在写入数据到 TWDR 寄存器时控制总线的暂停等。TWCR 寄存器还可以给出在 TWDR 寄存器无法访问期间, 试图将数据写入 TWDR 而引起的写入冲突信息, 表 8.3 是 TWCR 寄存器的内部结构。

表 8.3 TWI 控制寄存器 TWCR

BIT	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	Res	TWIE
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
初始值	0	0	0	0	0	0	0	0

- **TWINT**: TWI 总线中断标志。当 TWI 总线操作完时被置“1”, 若 SREG 的 I 标志以及 TWCR 寄存器的 TWIE 标志位也置“1”, 则产生 TWI 总线中断事件。当 TWINT 被置“1”时, SCL 信号的低电平被延长。中断服务子程序不会自动清零, 必须通过软件对 TWINT 标志写“1”来将该标志位清零。需要注意的是, 只要该标志位被清零, TWI 立即开始下一步工作, 因此在对 TWINT 清零之前一定要先完成对地址寄存器 TWAR、状态寄存器 TWSR, 以及数据寄存器 TWDR 的访问。
- **TWEA**: TWI 使能应答, TWEA 标志控制应答脉冲的产生。若 TWEA 被置位, 出现如下条件时接口发出 ACK 脉冲: ATmega16 的从机地址与主机发出的地址相符合、TWAR 的 TWGCE 置位时接收到广播呼叫、在主机/从机接收模式下接收到一个字节的数据。对 TWEA 位的清零可以使器件暂时脱离总线, 置位后 ATmega16 重新恢复地址识别。



- **TWSTA**: TWI 总线的 START 状态标志。当 ATmega16 希望自己成为总线上的主机时需要置位 TWSTA, TWI 硬件模块检测总线是否可用, 若总线空闲, ATmega16 就在总线上产生 START 状态; 若总线忙, ATmega16 就一直等到检测到一个 STOP 状态, 然后产生 START 以声明自己希望成为主机, 发送 START 之后必须用软件清零 TWSTA 位。
- **TWSTO**: TWI 总线 STOP 状态标志。在主机模式下, 如果置位 TWSTO 位, TWI 接口将在总线上产生 STOP 信号, 然后 TWSTO 位自动清零; 在从机模式下, 置位 TWSTO 可以使接口从错误状态恢复到未被寻址的状态, 此时总线上不会有 STOP 状态产生, 但 TWI 总线返回一个定义好的未被寻址的从机模式且释放 SCL 与 SDA 数据线为高阻态。
- **TWWC**: TWI 写冲突标志位。当 TWINT 为低时写数据寄存器 TWDR 将置位 TWWC 标志位; 当 TWINT 为高时, 每一次对 TWDR 的写访问都将更新此标志位。
- **TWEN**: TWI 使能位, TWEN 位用于使能 TWI 操作与激活 TWI 接口。当 TWEN 位被置“1”时, 外部引脚将切换到 SCL 与 SDA 引脚, 使能波形斜率限制器与尖峰滤波器。如果该位清零, TWI 模块将被关闭, 所有 TWI 传输将被终止。
- **Res**: 保留位, 当读取该位时返回值为“0”。
- **TWIE**: TWI 中断使能位。当 SREG 的 I 以及 TWIE 被置位时, 激活 TWI 总线中断。

### 8.2.3 TWI 状态寄存器 (TWSR)

TWI 的状态寄存器 (TWSR) 用于指示 TWI 模块的相关工作状态, 其内部结构如表 8.4 所示。

表 8.4 TWI 状态寄存器 TWSR

BIT	TWS7	TWS6	TWS5	TWS4	TWS3	Res	TWSP1	TWSP0
读/写	R	R	R	R	R	R	R/W	R/W
初始值	1	1	1	1	1	0	0	0

- **TWS7 ~ TWS3**: TWI 总线状态位, 此 5 位用来反映 TWI 的逻辑和总线的状态, 不同的状态代码会在 8.3 节介绍。



#### 注意

从 TWSR 位读出的值包括 5 位状态值与 2 位预分频值, 检测状态位时应屏蔽预分频位为“0”, 这使状态检测独立于预分频器设置。

- **Res**: 保留位, 当读该位时返回值为“0”。
- **TWPS1 ~ TWPS0**: TWI 总线的预分频位, 这两位可以进行读操作和写操作, 用于控制比特率预分频因子, 如表 8.5 所示。

表 8.5 TWI 总线的比特率预分频

TWPS1	TWPS0	预分频值
0	0	1

续表

TWPS1	TWPS0	预分频值
0	1	4
1	0	16
1	1	64

### 8.2.4 TWI 数据寄存器 (TWDR)

在 TWI 模块的发送模式，TWDR 寄存器中存放了要发送的字节；在接收模式，TWDR 存放了接收到的数据。

当 TWINT 位被置“1”时，TWI 模块没有进行移位工作，该寄存器是可写的。在第一次 TWI 总线中断发生之前，用户不能够初始化数据寄存器。当 TWINT 位被置位时，TWDR 寄存器的数据是稳定的。当数据移出时，TWI 总线上的数据同时移入寄存器，TWDR 寄存器中总是包含了总线上出现的最后一个字节，除非 ATmega16 是从掉电或省电模式被 TWI 总线中断唤醒，此时 TWDR 寄存器的内容是随机的。当 TWI 总线仲裁失败后，主机将切换为从机，但是总线上出现的数据不会丢失。ACK 的处理由 TWI 总线模块逻辑自动管理，ATmega16 不能直接访问 ACK 事件，该寄存器内部结构如表 8.6 所示。

表 8.6 TWI 数据寄存器 TWDR

BIT	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	1	1	1	1	1	1	1	1

### 8.2.5 TWI 从机地址寄存器 (TWAR)

TWAR 寄存器用于存放 TWI 从机地址，其内部结构如表 8.7 所示。

表 8.7 TWI 地址寄存器 TWAR

BIT	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	1	1	1	1	1	1	1	0

- TWA6 ~ TWA0：存放 TWI 从机地址。
- TWGCE：TWI 广播识别使能位，当该位被置“1”时，ATmega16 将能检测到 TWI 总线的广播数据。

## 8.3 TWI 总线模块的使用

ATmega16 的 TWI 接口是面向字节和基于中断的，即所有的总线事件，如接收到一个字节或发送了一个 START 信号等，就会产生一个 TWI 总线中断，由于 TWI 接口是基于中断的，因此 TWI 接口在字节发送和接收过程中，不需要进行软件的操作。TWCR 寄存器的

TWI 中断允许 TWIE 位和 SRE 寄存器的全局中断允许位一起决定了应用程序是否响应 TWINT 标志位产生的中断请求。如果 TWIE 位被清零，应用程序只能采用轮询 TWINT 标志位的方法来检测 TWI 总线状态。

当 TWINT 标志位被置“1”时，表示 TWI 接口已经完成了当前的操作，等待软件响应。在这种情况下，TWI 总线状态寄存器 TWSR 包含了表明当前 TWI 总线状态的值，用户软件可以读取 TWSR 的状态码以判别此时的状态是否正确，并通过设置 TWCR 与 TWDR 寄存器，以决定在下一个 TWI 总线周期 TWI 模块应该如何工作。

图 8.4 是 ATmega16 的 TWI 总线数据传递过程示意图，在该示意图中，主机发送一个数据字节给从机，其详细描述如下。

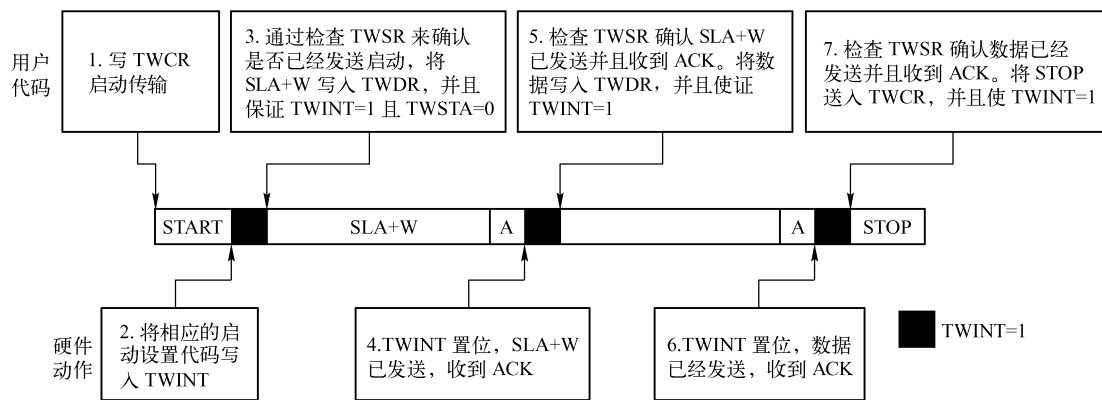


图 8.4 ATmega16 的 TWI 总线数据传递过程

(1) 向 TWCR 寄存器写入特定值，发送 START 信号，在写入时 TWINT 位会被置位，需要通过向 TWINT 位写“1”以清除此标志。在 TWCR 寄存器的 TWINT 的置位期间，TWI 总线不会启动任何操作。一旦 TWINT 被清零，TWI 总线由 START 信号启动一次数据传输。

(2) 在发送 START 信号后，TWCR 寄存器的 TWINT 标志位置位，TWCR 更新为新的状态码，表示 START 信号成功发送。

(3) 用户的应用代码应该通过检验 TWSR 来确定 START 信号已成功发送。如果 TWSR 显示为其他，软件可以执行一些指定操作，如调用错误处理程序。如果状态码与预期的一致，应用程序必须将 SLA + W 输入 TWDR，TWDR 可同时在地址与数据中使用。TWDR 载入 SLA + W 后，TWCR 必须写入特定值指示 TWI 硬件发送 SLA + W 信号。写入的值将在后面说明。在写入值时 TWINT 位要置位，这非常重要。给 TWINT 写入“1”，清除此标志。TWCR 寄存器的 TWINT 置位，TWI 总线不会启动任何操作。一旦 TWINT 被清零，TWI 将启动地址包的传送。

(4) 在地址包发送后，TWCR 寄存器的 TWINT 的标志位被置“1”，TWDR 被更新为新的状态码以表示地址包成功发送，状态码还会反映从机是否响应包。

(5) 软件通过检查 TWSR 寄存器以确定地址包已成功发送，并且 ACK 已经收到，如果 TWSR 显示为未完成，软件可以执行一些指定操作，如调用错误处理程序等。如果状态码与预期的一致，应用程序必须将数据包送入 TWDR，接下来 TWCR 必须写入特定值指示 TWI

模块发送 TWDR 中的数据包。在写入值时, TWINT 位必须被置“1”, 在 TWCR 寄存器中的 TWINT 被置位期间 TWI 不会启动任何操作, 一旦被清零, TWI 总线启动数据包的传输。

(6) 在数据包发送后, TWCR 寄存器的 TWINT 标志位被置“1”, TWSR 更新为新的状态码, 以表示数据包成功发送, 状态代码还会反映从机是否对包进行响应。

(7) 软件通过检查 TWSR 寄存器以确定地址包已成功发送并且 ACK 返回, 如果 TWSR 寄存器显示为其他, 软件可能执行一些指定操作, 如调用错误处理程序。如果状态码与预期的一致, TWCR 寄存器必须写入特定值指示 TWI 模块发送 STOP 信号。在写入值时, TWINT 位要置“1”, 这非常重要, 必须向 TWINT 位写入“1”以清除此标志, TWCR 寄存器中的 TWINT 置位期间 TWI 不会启动任何操作。一旦 TWINT 位被清零, TWI 总线启动 STOP 信号的传送, 需要注意的是, TWINT 在 STOP 位状态发送后不会置位。

(8) 当 TWI 模块完成一次操作并等待反馈时, TWINT 位被置“1”, 直到 TWINT 被清零, 时钟线 SCL 才会拉低。

(9) 当 TWINT 标志被置“1”时, 用户必须用与下一个 TWI 总线周期相关的值更新 TWI 的寄存器。例如, TWDR 寄存器必须载入下一个总线周期中要发送的值。

(10) 当所有的 TWI 寄存器得到更新, 而且其他挂起的应用程序也已经结束时, TWCR 寄存器被写入数据。当写 TWCR 时, TWINT 应该被置“1”, 必须通过软件对 TWINT 写“1”以清除此标志, TWI 模块将开始执行由 TWCR 寄存器设定的操作。

## 8.4 TWI 总线模块的数据传输方式

ATmega16 的 TWI 总线模块提供了主机发送 (MT)、主机接收 (MR)、从机发送 (ST) 和从机接收 (SR) 四种不同的工作模式。同一个应用程序可以使用 ATmega16 的多个工作模式。例如, ATmega16 可以使用 MT 模式给 TWI 接口的 E<sup>2</sup>PROM 写入数据, 用 MR 模式从 E<sup>2</sup>PROM 读取数据。如果 TWI 总线上有其他主机存在, 则它们可能给 ATmega16 发送数据, 此时就可以使用 SR 工作模式。

### 8.4.1 主机发送模式 (MT)

在主机发送模式下, ATmega16 可以向 TWI 总线上的从机发送数据, 其发送模式状态码如表 8.8 所示。

表 8.8 主机发送模式状态码

状态码 (TWSR) 预分频 = 0	TWI 总线状态	代码操作					下一步操作
		读/写 TWDR	对 TWCR 的操作				
			STA	STO	TWINT	TWEA	
0x08	START 已发送	SLA + W	0	0	1	X	将发送 SLR + W 收到 ACK 或者 NOTACK
0x11	重复 START 已经发送	SLA + W 或者 SLA + R	0 0	0 0	1 1	X X	将发送 SLR + W 收到 ACK 或者 NOTACK 将发送 SLR + R 收到 ACK 或者 NOTACK

续表

状态码 (TWSR) 预分频 = 0	TWI 总线状态	代码操作					下一步操作
		读/写 TWDR	对 TWCR 的操作				
			STA	STO	TWINT	TWEA	
0x18	SLA + W 已经 发送 接收到 ACK	加载数据 或者不操作 TWDR	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	发送数据, 接收 ACK 或者 NOTACK 发送重复 START 发送 STOP, TWSTO 将复位 发送 STOP, 然后发送 START, TWSTO 将复位
0x20	SLA + W 已发送 接收到 NOTACK	加载数据 或者不操作 TWDR	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	发送数据, 接收 ACK 或者 NOTACK 发送重复 START 发送 STOP, TWSTO 将复位 发送 STOP, 然后发送 START, TWSTO 将复位
0x28	数据已经发送 接收到 ACK	加载数据 或者不操作 TWDR	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	将发送数据, 接收 ACK 或者 NOTACK 发送重复 START 发送 STOP, TWSTO 将复位 将发送 STOP, 然后发送 START, TWSTO 将复位
0x30	数据已经发送 接收到 NOTACK	加载数据 或者不操作 TWDR	0 1 0 1	0 0 1 1	1 1 1 1	X X X X	将发送数据, 接收 ACK 或者 NOTACK 发送重复 START 发送 STOP, TWSTO 将复位 将发送 STOP, 然后发送 START, TWSTO 将复位
0x38	SLA + W 或者数据仲裁失败	不操作 TWDR	0 1	0 0	1 1	X X	TWI 总线将被释放, 并且进入未寻址从机模式 总线空闲后发送 START

## 8.4.2 主机接收模式 (MR)

在主机接收模式下, ATmega16 可以从 TWI 总线上的从机接收数据, 其接收模式状态码如表 8.9 所示。

表 8.9 主机接收模式状态码

状态码 (TWSR) 预分频 = 0	TWI 总线状态	代码操作					下一步操作
		读/写 TWDR	对 TWCR 的操作				
			STA	STO	TWINT	TWEA	
0x08	START 已发送	SLA + R	0	0	1	X	将发送 SLR + R 收到 ACK 或者 NOTACK



续表

状态码 (TWSR) 预分频 = 0	TWI 总线状态	代码操作					下一步操作
		读/写 TWDR	对 TWCN 的操作				
			STA	STO	TWINT	TWEA	
0x11	重复 START 已经发送	SLA + R 或者 SLA + W	0 0	0 0	1 1	X X	将发送 SLR + R 收到 ACK 或者 NOTACK 将发送 SLR + W 收到 ACK 或者 NOTACK
0x38	SLA + R 或者 NO-TACK 仲裁失败	不操作 TWDR	0 1	0 0	1 1	X X	2 线串行总线被释放, 并进入未寻址从机模式, 等待总线空闲后发送 START
0x40	SLA + R 已发送 接收到 ACK	不操作 TWDR	0 0	0 0	1 1	0 1	接收数据, 返回 NOTACK 接收数据, 返回 ACK
0x48	SLA + R 已发送 接收到 NOTACK	不操作 TWDR	1 0 1	0 1 1	1 1 1	X X X	重复发送 START 发送 STOP, TWSTO 复位 发 送 STOP, 然 后 发 送 START, TWSTO 将复位
0x50	接收到数据 ACK 已返回	读数据	0 0	0 0	1 1	0 1	接收数据, 返回 NOTACK 接收数据, 返回 ACK
0x58	接收到数据 NOTACK 已 经 返回	读数据	1 0 1	0 1 1	1 1 1	X X X	发送 START 发送 STOP, TWSTO 有复位 发 送 STOP, 然 后 发 送 START, TWSTO 将复位

### 8.4.3 从机发送模式 (ST)

在从机发送模式下, ATmega16 可以向 TWI 总线上的主机发送数据, 其状态编码如表 8.10 所示。

表 8.10 从机发送模式状态码

状态码 (TWSR) 预分频 = 0	TWI 总线状态	代码操作					下一步操作
		读/写 TWDR	对 TWCR 的操作				
			STA	STO	TWINT	TWEA	
0x08	START 已发送	SLA + R	0	0	1	X	将发送 SLR + R 收到 ACK 或者 NOTACK
0x11	重复 START 已经 发送	SLA + R 或者	0	0	1	X	将发送 SLR + R 收到 ACK 或者 NOTACK
		SLA + W	0	0	1	X	将发送 SLR + W 收到 ACK 或者 NOTACK
0x38	SLA + R 或者 NO-TACK 仲裁失败	不操作 TWDR	0 1	0 0	1 1	X X	2 线串行总线被释放, 并进入未寻址从机模式, 等待总线空闲后发送 START
0x40	SLA + R 已发送 接收到 ACK	不操作 TWDR	0 0	0 0	1 1	0 1	接收数据, 返回 NOTACK 接收数据, 返回 ACK

续表

状态码 (TWSR) 预分频 = 0	TWI 总线状态	代码操作					下一步操作
		读/写 TWDR	对 TWCR 的操作				
			STA	STO	TWINT	TWEA	
0x48	SLA + R 已发送 接收到 NOTACK	不操作 TWDR	1 0 1	0 1 1	1 1 1	X X X	重复发送 START 发送 STOP, TWSTO 复位 发 送 STOP, 然 后 发 送 START, TWSTO 将复位
0x50	接收到数据 ACK 已返回	读数据	0 0	0 0	1 1	0 1	接收数据, 返回 NOTACK 接收数据, 返回 ACK
0x58	接收到数据 NOTACK 已 经 返回	读数据	1 0 1	0 1 1	1 1 1	X X X	发送 START 发送 STOP, TWSTO 有复位 发 送 STOP, 然 后 发 送 START, TWSTO 将复位

#### 8.4.4 从机接收模式 (SR)

在从机接收模式下, ATmega16 作为从机从 TWI 总线上接收数据, 其状态编码如表 8.11 所示。

表 8.11 从机接收模式状态码

状态码 (TWSR) 预分频 = 0	TWI 总线状态	代码操作					下一步操作
		读/写 TWDR	对 TWCR 的操作				
			STA	STO	TWINT	TWEA	
0x08	START 已发送	SLA + R	0	0	1	X	将发送 SLR + R 收到 ACK 或者 NOTACK
0x11	重复 START 已经 发送	SLA + R 或者	0	0	1	X	将发送 SLR + R 收到 ACK 或者 NOTACK
		SLA + W	0	0	1	X	将发送 SLR + W 收到 ACK 或者 NOTACK
0x38	SLA + R 或者 NO- TACK 仲裁失败	不操作 TWDR	0 1	0 0	1 1	X X	2 线串行总线被释放, 并进 入未寻址从机模式, 等待总线 空闲后发送 START
0x40	SLA + R 已发送 接收到 ACK	不操作 TWDR	0 0	0 0	1 1	0 1	接收数据, 返回 NOTACK 接收数据, 返回 ACK
0x48	SLA + R 已发送 接收到 NOTACK	不操作 TWDR	1 0 1	0 1 1	1 1 1	X X X	重复发送 START 发送 STOP, TWSTO 复位 发 送 STOP, 然 后 发 送 START, TWSTO 将复位
0x50	接收到数据 ACK 已返回	读数据	0 0	0 0	1 1	0 1	接收数据, 返回 NOTACK 接收数据, 返回 ACK
0x58	接收到数据 NOTACK 已 经 返回	读数据	1 0 1	0 1 1	1 1 1	X X X	发送 START 发送 STOP, TWSTO 有复位 发 送 STOP, 然 后 发 送 START, TWSTO 将复位



## 8.5 TWI 总线的仲裁

如果有多个主机连接在同一条 TWI 总线上，它们中的一个或多个也许会同时开始一个数据传送。TWI 总线协议确保在这种情况下，通过一个仲裁过程，允许其中的一个主机进行传送而不会丢失数据，图 8.5 为多主机仲裁的示意图。

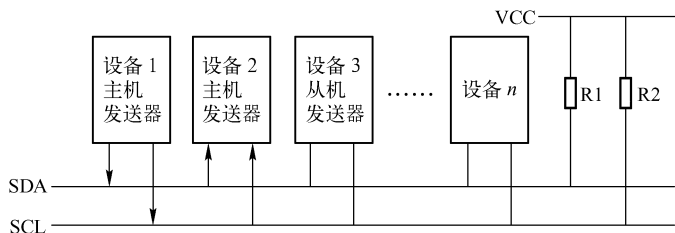


图 8.5 多主机仲裁

有如下几种的情况会产生总线仲裁过程。

- 两个或更多的主机同时与一个从机进行通信。在这种情况下，无论主机或从机都不知道有总线的竞争。
- 两个或更多的主机同时对同一个从机进行不同的数据或方向的访问。在这种情况下，会在 READ/WRITE 位或数据间发生仲裁。当主机试图在 SDA 数据线上输出一个高电平时，如果其他主机已经输出低电平，则该主机在总线仲裁中失败。失败的主机将转换成未被寻址的从机模式，或等待总线空闲后发送一个新的 START 信号，这由用户程序决定。
- 两个或更多的主机访问不同的从机。在这种情况下，总线仲裁在 SLA 数据线上发生，当主机试图在 SDA 线上输出一个高电平时，如有其他主机已经输出低电平，则该主机将在总线仲裁中失败。在 SLA 总线仲裁失败的主机将切换到从机模式，并检查自己是否被获得总线控制权的主机寻址。如果被寻址，它将进入 SR 或 ST 模式，这取决于 SLA 的 READ/WRITE 位的值。如果它未被寻址，将转换到未被寻址的从机模式或等待总线空闲，发送一个新的 START 信号，这由用户程序决定。

## 8.6 SPI 总线基础

除了 TWI (I<sup>2</sup>C) 总线接口模块，ATmega16 还内置了一个 SPI (Serial Peripheral Interface) 接口总线模块，它使得 ATmega16 可以和各种外围设备以串行方式进行最高能达到 3Mb/s 的速度的数据通信，其由内置的控制单元以及四根外部引脚组成，

ATmega16 的 SPI 总线接口占用了 4 根 I/O 引脚来作为数据通信的信号线，其分别定义如下：

- MISO (PB3)：主入从出数据线，主机的数据输入线，从机的数据输出线
- MOSI (PB2)：主出从入数据线，主机的数据输出线，从机的数据输入线
- SCK (PB1)：串行时钟数据线，由主机发出，对于从机是输入信号，当主机发起一次

传送时, 自动发出 8 个 SCK 信号, 数据移位发生在 SCK 的每一次跳变上。

- SS (PB0): 外设片选数据线, 数据传送开始前允许从机 SPI 接口工作的片选信号线。

和 TWI (I<sup>2</sup>C) 总线不同, 在 SPI 总线上只允许存在一个主机, 但是可以存在多个从机, 主机使用 SS 信号线来选择从机, 在时钟信号 SCK 的上升/下降沿主机数据从主机的 MOSI 引脚发送给被 SS 选中的从机的 MISO 引脚, 而在下一次下降/上升沿上从机数据从从机的 MISO 引脚上返回到主机的 MOSI 引脚上, SPI 总线的工作过程类似一个 16 位的移位寄存器, 其中 8 位数据在主机中, 另外的 8 位数据在从机中, ATmega16 使用 SPI 总线接口来扩展外围器件的电路如图 8.6 所示。

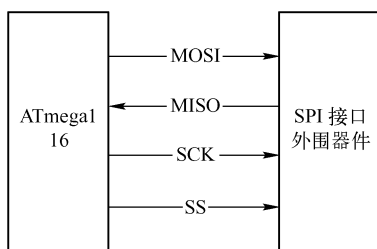


图 8.6 ATmega16 的 SPI 总线接口

和 TWI (I<sup>2</sup>C) 总线相同, SPI 总线的数据交换过程也需要时钟驱动, SPI 总线的时钟有时钟极性 (CPOL) 和时钟相位 (CPHA) 两个参数, 前者决定了有效时钟是高电平还是低电平, 后者决定有效时钟的相位, 这两个参数配合起来决定了 SPI 总线的数据时序, 如图 8.7 和图 8.8 所示。

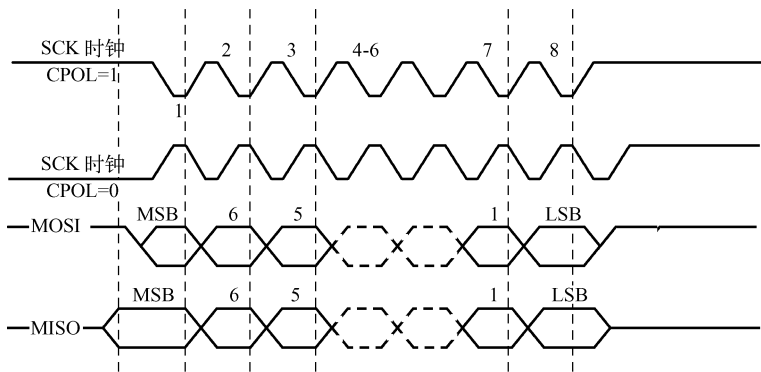


图 8.7 CPHA = 0 时的 SPI 总线数据传输时序

从图 8.7 和图 8.8 中可以看到:

- 如果 CPOL = 0, 串行同步时钟的空闲状态为低电平;
- 如果 CPOL = 1, 串行同步时钟的空闲状态为高电平;
- 如果 CPHA = 0, 在串行同步时钟的第一个跳变沿 (上升或下降) 采样数据;
- 如果 CPHA = 1, 在串行同步时钟的第二个跳变沿 (上升或下降) 采样数据。

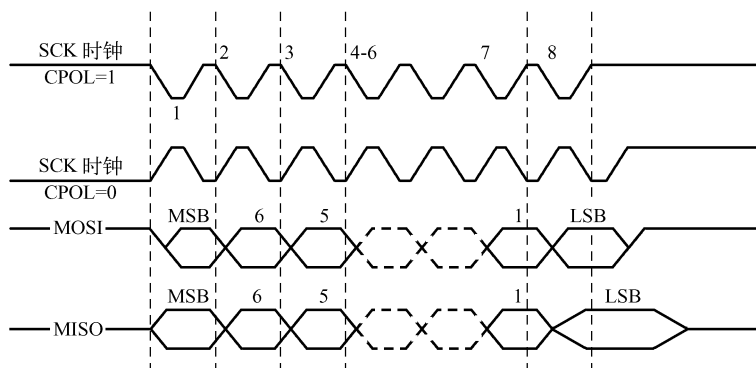


图 8.8 CPHA = 1 时的 SPI 总线数据传输时序

## 8.7 SPI 总线模块相关寄存器

ATmega16 同样通过相关寄存器的操作来实现对 SPI 接口的控制，这些寄存器包括 SPI 控制寄存器 SPCR、SPI 状态寄存器 SPSR 和 SPI 数据寄存器 SPDR。

### 8.7.1 SPI 控制寄存器 SPCR

SPI 总线接口的控制寄存器 SPCR 用于对 SPI 总线接口进行设置和操控，其内部结构如表 8.12 所示。

表 8.12 SPI 控制寄存器 SPCR

BIT	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- SPIE：SPI 中断使能位。当该位被置位，如果 SPSR 寄存器的 SPIF 位和 SREG 寄存器的全局中断使能位 I 被置位，就会触发 SPI 中断事件。
- SPE：SPI 使能位。当该位被置位时将使能 SPI 总线接口，在进行任何 SPI 操作之前必须置位 SPE 位。
- DORD：数据次序选择位。当 DORD 被置“1”时，首先发送数据的最低位（LSB），否则数据的最高位（MSB）首先被发送。
- MSTR：主/从选择位。当 MSTR 被置位时选择主机工作模式，否则为从机工作模式，如果 MSTR 为“1”时，SS 引脚配置为输入且被拉低，则 MSTR 被清零，寄存器 SPSR 的 SPIF 位被置位，用户必须重新设置 MSTR 使 SPI 总线重新进入主机模式。
- CPOL：时钟极性选择位。CPOL 被置位表示 SPI 总线空闲时 SCK 为高电平，否则 SPI 总线空闲时 SCK 为低电平，参考图 8.7 与图 8.8，CPOL 控制功能如表 8.13 所示。



表 8.13 CPOL 控制功能

CPOL	起 始 沿	结 束 沿
0	上升沿	下降沿
1	下降沿	上升沿

- CPHA：时钟相选择位。CPHA 位决定是在 SCK 的起始沿对数据采样还是在 SCK 的结束沿对数据进行采样，参考图 8.7 与图 8.8，CPHA 控制功能如表 8.14 所示。

表 8.14 CPHA 控制功能

CPHA	起 始 沿	结 束 沿
0	采样	采样
1	设置	设置

- SPR1、SPR0：SPI 时钟速率选择位。这两位用于联合确定主机的 SCK 时钟速率，SPR1 和 SPR0 位对从机没有影响，SCK 时钟频率和 ATmega16 的工作时钟频率 $f_{osc}$ 的关系如表 8.15 所示。

表 8.15 SCK 时钟频率设置

SPI2X	SPR1	SPR0	SCK 频率
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

### 8.7.2 SPI 状态寄存器 SPSR

SPI 状态寄存器 SPSR 用于反映 SPI 总线接口的工作状态，其内部结构如表 8.16 所示。

表 8.16 SPI 状态寄存器 SPSR

BIT	SPIF	WCOL	—	—	—	—	—	SPI2X
读/写	R	R	R	R	R	R	R	R/W
初始值	0	0	0	0	0	0	0	0

- SPIF：SPI 总线中断标志位。当 SPI 的串行发送结束后，SPIF 将被置“1”，若此时寄存器 SPCR 的 SPIE 和全局中断使能位 I 被置位，则产生一个 SPI 中断事件。如果 SPI 总线接口为主机工作模式，当 SS 引脚配置为输入且被拉低，SPIF 也将被置“1”触

发中断事件，进入中断服务程序后，SPIF 将自动清零，或者用户程序可以通过先读 SPSR 寄存器，紧接着访问 SPDR 寄存器的方式来对 SPIF 位清零。

- WCOL: SPI 总线接口写冲突标志位。在数据发送过程中，对 SPI 总线接口数据寄存器 SPDR 写数据时将置位 WCOL，该位可以通过先读 SPSR 寄存器，紧接着访问 SPDR 来清零。
- SPI2X: SPI 倍速控制位。当该位被置“1”后，SPI 总线的速度加倍，若工作在主机工作模式下，则 SCK 时钟的频率可达到 ATmega16 工作频率的 1/2，若工作在从机工作模式，则只能达到 ATmega16 的工作频率 1/4，参考表 8.15。

### 8.7.3 SPI 数据寄存器 SPDR

SPI 数据寄存器 SPDR 为读/写寄存器，用于存放 SPI 总线接口需要传输的数据，对寄存器的写入操作将启动数据传送，对寄存器的读操作将读取接收缓冲器中的数据，该寄存器的内部位结构如表 8.17 所示。

表 8.17 SPI 数据寄存器 SPDR

BIT	MSB							SPI2X
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	X	X	X	X	X	X	X	X

## 8.8 SPI 总线接口的工作模式

SPI 总线的主机和从机之间的连接如图 8.9 所示，该系统中包括两个移位寄存器和一个 SPI 时钟发生器。通过将需要通信的从机的 SS 引脚拉低，主机启动一次通信过程，主机和从机将需要发送的数据放入相应的移位寄存器。主机在 SCK 引脚上产生时钟脉冲以交换数据。主机的数据从主机的 MOSI 引脚上移出，从从机的 MOSI 引脚移入；从机的数据由从机的 MISO 引脚移出，从主机的 MISO 引脚移入。主机通过将从机的 SS 拉高实现与从机的同步。

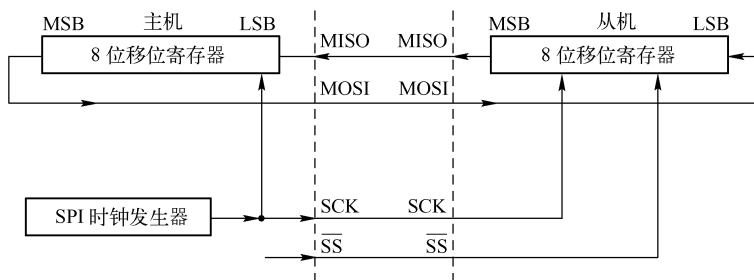


图 8.9 SPI 的主机从机数据交互方式

当 ATmega16 被配置为 SPI 主机时，SPI 总线接口并不自动控制 SS 引脚，必须由用户代码来控制，当对 SPI 数据寄存器写入数据即启动 SPI 时钟时，将 8 位的数据移入从机。传输

结束后 SPI 时钟停止，传输结束标志 SPIF 标志位被置“1”。如果此时 SPCR 寄存器的 SPI 中断使能位 SPIE 被置“1”，即触发中断事件，此时主机可以继续往 SPDR 寄存器写入待发送的数据或者是将从机的 SS 引脚拉高以通知数据包发送完成，最后进来的数据将一直保存在缓冲寄存器里。

当 ATmega16 被配置为 SPI 从机时，只要 SS 引脚为高，SPI 总线接口将一直保持睡眠状态，并保持 MISO 为三态状态，在此状态下，用户的软件可以更新 SPI 数据寄存器 SPDR 的内容。即使此时 SCK 引脚有输入时钟，SPDR 寄存器的数据也不会移出，直至 SS 引脚被拉低。当整个字节完全移出之后，传输结束标志位 SPIF 被置“1”。如果此时 SPCR 寄存器的 SPI 中断使能位 SPIE 置位，就会产生中断请求。在读取移入的数据之前，从机可以继续往 SPDR 寄存器里写入数据，最后进来的数据将一直保存在缓冲寄存器里。

ATmega16 的 SPI 总线的发送方向只有一个缓冲器，而在接收方向有两个缓冲器，所以在发送时一定要等到移位过程全部结束后才能对 SPI 数据寄存器写入数据；而在接收数据时，则需要在下一个字符移位过程结束之前，通过从 SPI 数据寄存器将当前接收到的字符取走，否则前一个字节数据将丢失。

在 ATmega16 工作于从机模式时，控制逻辑对 SCK 引脚的输入信号进行采样，为了保证对时钟信号的正确采样，SPI 时钟频率不能超过  $f_{osc}/4$ 。

在 ATmega16 的 SPI 总线接口被使能后，MOSI、MISO、SCK 和 SS 引脚的数据方向将按照表 8.18 所示自动进行配置。

表 8.18 SPI 引脚配置

引 脚	SPI 主机方向	SPI 从机方向
MOSI	用户自定义	输入
MISO	输入	用户自定义
SCK	用户自定义	输入
SS	用户自定义	输入

当 ATmega16 的 SPI 总线接口工作在从机模式时，其从机选择引脚 SS 总是被配置为输入状态，如果 SS 引脚上加一个低电平，将激活 ATmega16 的 SPI 总线接口，MISO 引脚被用户主动配置为输出状态，而其他引脚成为输入状态；当 SS 为高电平时，所有的引脚将进入输入状态，SPI 总线逻辑复位，不再接收数据。

SS 引脚对于数据包或者字节的同步非常有用，可以使从机的位计数器与主机的时钟发生器同步。当 SS 被拉高时，SPI 从机立即复位接收和发送逻辑，并丢弃移位寄存器里不完整的数据。

当 ATmega16 工作于主机模式时，用户可以自定义 SS 引脚的输入/输出方向。若 SS 引脚被配置为输出引脚，则此引脚可以用作普通的 I/O 口而不影响 SPI 总线接口，其典型应用是用来驱动从机的 SS 引脚。如果 SS 引脚配置为输入，则必须保持为高电平以保证 SPI 总线的正常工作。如果 ATmega16 工作在主机模式，SS 引脚为输入状态，但被外设拉低，则 SPI 总线接口会将此低电平看作有一个外部主机将自己选择为从机，为了防止总线冲突，SPI 总线接口将进行如下动作。



- 清零 SPCR 寄存器的 MSTR 位，使 ATmega16 的 SPI 总线接口进入从机工作模式，MOSI 引脚和 SCK 引脚进入输入工作状态。
- 置位 SPSR 寄存器的 SPIF 位，此时若 SPI 中断和全局中断开放，则会触发一个中断事件，进入中断服务子程序。



### 注意

在 ATmega16 使用中断方式处理 SPI 总线接口的数据传输，并且存在 SS 引脚被拉低的可能性时，中断服务程序应检查 MSTR 位是否为“1”，若为“0”，用户必须将其置“1”，以重新使能 SPI 总线接口的主机模式。

## 8.9 TWI 和 SPI 总线模块应用实例

### 8.9.1 ATmega16 双机使用 TWI 总线模块进行通信

本应用是两片 ATmega16 使用 TWI 总线接口进行通信的实例，单片机 A（U1）通过 TWI 总线接口将数据发送到单片机 B（U2），单片机 B 接收到数据之后将其从串口送出。

#### 1. 实例的设计思路

单片机 A 作为主机发送，单片机 B 作为从机接收数据，ATmega16 的 TWI 总线接口作为主机发送器的操作步骤如表 8.8 所示，ATmega16 的 TWI 总线接口作为从机接收器的操作步骤如表 8.9 所示。

#### 2. 实例的 Proteus 电路

实例 Proteus 电路如图 8.10 所示，单片机 A 和单片机 B 通过 SCL（PC0）引脚和 SDA（PC1）引脚相连，一个 I<sup>2</sup>C 总线调试器（I<sup>2</sup>C DEBUGGER）加在 TWI 总线上用于监测数据状态，一个虚拟终端用于接收单片机 B 的串口输出，实例涉及的 Proteus 器件如表 8.19 所示。

表 8.19 应用实例器件列表

器件名称	大类库	子类库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体

#### 3. 实例的应用代码

实例的应用代码如例 8.1 和例 8.2 所示。

代码首先对 TWI（I<sup>2</sup>C）总线模块进行初始化，然后调用 SendByte 函数来发送数据，在每次发送数据之间有 100ms 的延时。

#### 【例 8.1】TWI 总线双机通信主机代码

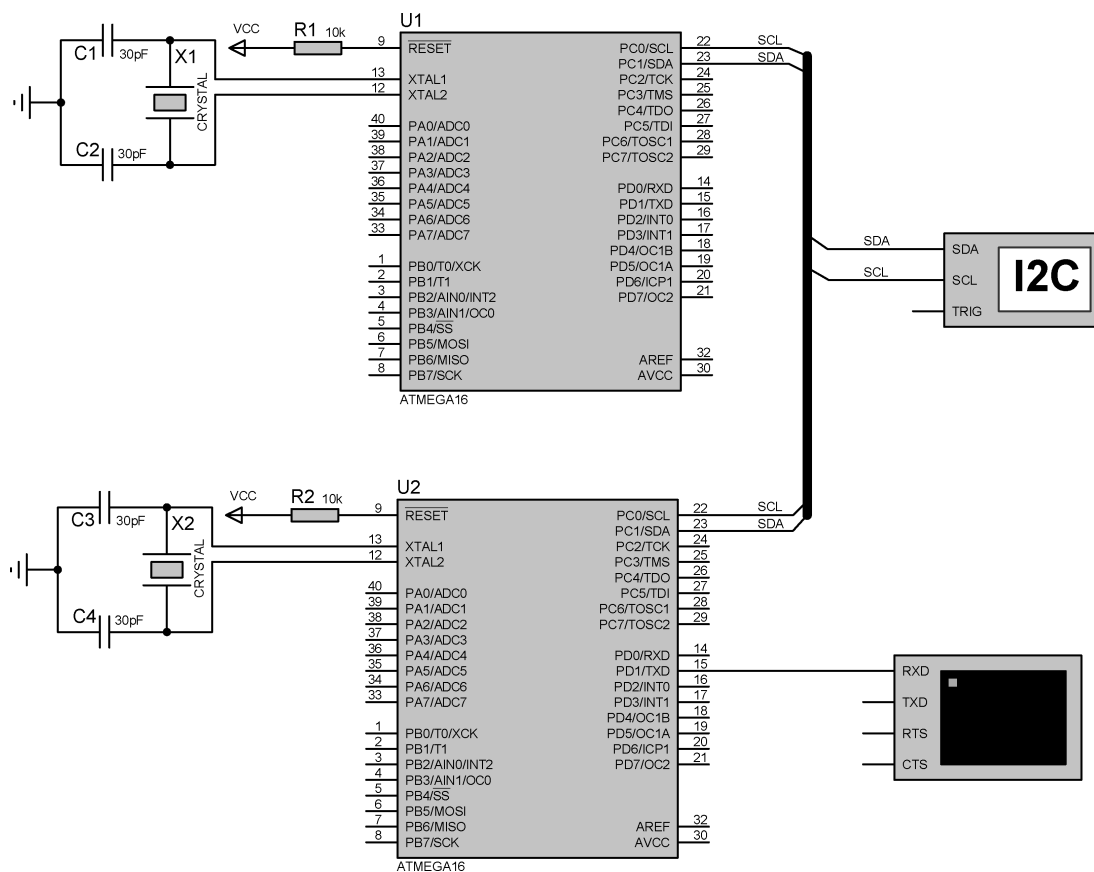


图 8.10 实例的 Proteus 电路

```
#include <iom16v.h>
```

```
#include <macros.h>
```

```
//TWI 总线状态定义
```

```
#define STATUS (TWSR&0x08)
```

```
#define SLA_W 0x32
```

```
#define SLA_R 0x33
```

```
#define SLAVER_ADDRESS 0x32
```

```
#define START 0x08
```

```
#define RE_START 0x10
```

```
#define MT_SLA_WRITE_ACK 0x18
```

```
#define MT_SLA_READ_ACK 0x40
```

```
#define MT_DATA_ACK 0x28
```

```
#define MT_READ_ACK 0x58
```

```
#define TW_MT_DATA_NACK 0x30
```

```
#define TRUE 1
```

```
#define FALSE 0
```



```

#define ALLLED      0xFF

void Delayus( unsigned int US)
{
    unsigned int i;
    US = US * 5/4;          //5/4 是在 8MHz 晶振下，通过软件仿真反复实验得到的数值
    for( i = 0; i < US; i ++ );
}
void Delayms( unsigned int MS)
{
    unsigned int i,j;
    for( i = 0; i < MS; i ++ )
        for( j = 0; j < 1141; j ++ ); //1141 是在 8MHz 晶振下，通过软件仿真反复实验得到的数值
}
//端口初始
void port_init( void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0xFF;
    DDRC  = 0xFF;
    PORTD = 0x00;
    DDRD  = 0x00;
}
//TWI 初始化函数
void twi_init( void)
{
    TWCR = ( 1 << TWEA ) | ( 1 << TWEN );    //主机模式，使能 TWI
    TWBR = 32;                                //波特率 100k 8M 晶振
}
//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    twi_init();
    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}

```

```

}
//启动 TWI
void Start( void)
{
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);
    while( ! (TWCR&(1 << TWINT))) ;
}
//停止 TWI
void Stop( void)
{
    TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
}
//写 8 位数据
void Write8bit(unsigned char i)
{
    TWDR = i;
    TWCR = (1 << TWINT) | (1 << TWEN);
    while( ! (TWCR&(1 << TWINT))) ;
}
/* 写一个字节 data, 成功则返回 1, 否则返回 0 */
unsigned char SendByte(unsigned char data)
{
    //启动 I2C
    Start();
    if( STATUS!= START) return 0;
    //发送从机地址 (写), 等待回应, 错误检测
    Write8bit(0);
    if( STATUS!= MT_SLA_WRITE_ACK) return 0;
    //发送数据, 等待回应, 错误检测
    Write8bit( data);
    if( STATUS!= MT_DATA_ACK) return 0;
    Stop();
    return 1; //发送成功, 返回 1
}
//主函数
void main( void)
{
    unsigned char i=0;
    init_devices();
    Delays(1);
    while(1)
    {

```



```

        for( i = 255; i > 0; i -- )
        {
            SendByte(i);
            Delays( 100 );
        }
    }
}

```

从机接收采用查询方式,当启动接收之后则一直等待接收,当接收到数据之后判断 TWSR 寄存器的状态,如果状态正确,则把接收到的数据通过 USART0 送出。

### 【例 8.2】 TWI 总线双机通信从机代码

```

#include < iom16v. h >
#include < macros. h >

#define STATUS                ( TWSR&0xf8 )
#define SLA_W                 0x32
#define SLA_R                 0x33
#define TWI_ADDRESS          0x32
#define START                 0x08
#define RE_START              0x10
#define MT_SLA_WRITE_ACK      0x18
#define MT_SLA_READ_ACK       0x40
#define MT_DATA_ACK           0x28
#define MT_READ_ACK           0x58
#define TW_MT_DATA_NACK       0x30
#define TRUE                  1
#define FALSE                 0
#define ALLLED                0xFF

//putchar 串口发送函数
extern void putchar( char inputdata )
{
    while( ! ( UCSRA & ( 1 << UDRE ) ) );
    UDR = inputdata;
}

//端口初始化函数
void port_init( void )
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
}

```

```

    PORTC = 0xFF;
    DDRC  = 0xFF;
    PORTD = 0x00;
    DDRD  = 0x00;
}
//TWI 总线模块初始化函数
void twi_init( void)
{
    //TWI 接口初始化, 从器件模式
    TWAR = TWI_ADDRESS | (1 << TWGCE);
    TWCR = (1 << TWEA) | (1 << TWEN) | (1 << TWIE);
    asm( "SEI" );
}
//UART 初始化函数, 9600b/s
void uart0_init( void)
{
    UCSRB = 0x00;
    UCSRA = 0x00;
    UCSRC = BIT( URSEL ) | 0x06;
    UBRRL = 0x33;
    UBRRH = 0x00;
    UCSRB = 0x48;
}
//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    uart0_init();
    twi_init();
    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主函数
void main( void)
{
    unsigned char temp, temp1;
    init_devices();
    while(1)

```

```

    {
        TWCR = (1 << TWINT) | (1 << TWEA) | (1 << TWEN);    //启动一次接收
        while((TWCR & (1 << TWINT)) == 0);    //等待有数据被接收
        temp = STATUS;
        if( temp == 0x90)
        {
            temp1 = TWDR;
            putchar(temp1);
        }
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，在单片机 B 的 TXD0 引脚上连接一个虚拟终端，可以看到对应的数据输出，如图 8.11 所示。

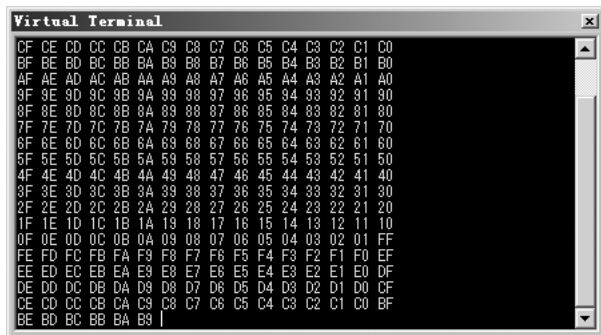


图 8.11 实例的仿真运行结果



### 总结

读者可以在单片机 A 上添加对应的串行模块驱动代码，使得单片机 A 可以从 PC 接收数据，然后通过 TWI 总线发送出去，这样就构成了一个 UART-TWI 总线桥。

#### 5. Proteus 中的 I<sup>2</sup>C DEBUGGER

I<sup>2</sup>C 总线调试器 (I<sup>2</sup>C DEBUGGER) 是用来观察当前 TWI (I<sup>2</sup>C) 总线数据的虚拟仪器。在 Proteus 中，单击工具箱中的“Virtual Instrument Mode”按钮，此时当前窗口会出现包括 I<sup>2</sup>C 总线调试器的所有虚拟仪器的列表，在该列表中选择 I<sup>2</sup>C 总线调试器后，在电路图中点击即可放置，如图 8.12 所示，其引脚说明如下。

- SDA：双向数据线。
- SCL：双向输入端，连接时钟。
- TRIG：触发输入，能引起存储序列被连续地放置到输出队列中。

双击 I<sup>2</sup>C 总线调试器，弹出如图 8.13 的属性设置对话框，其中各个主要参数说明如下。

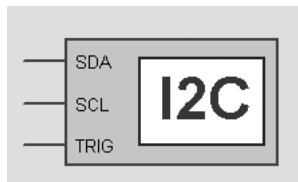
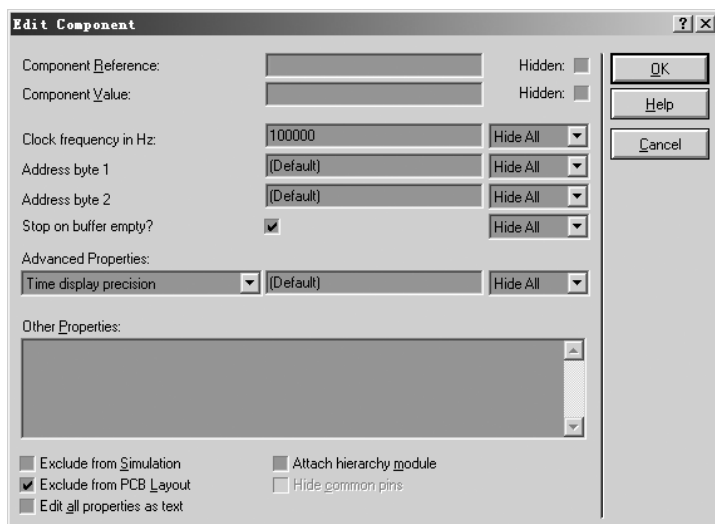
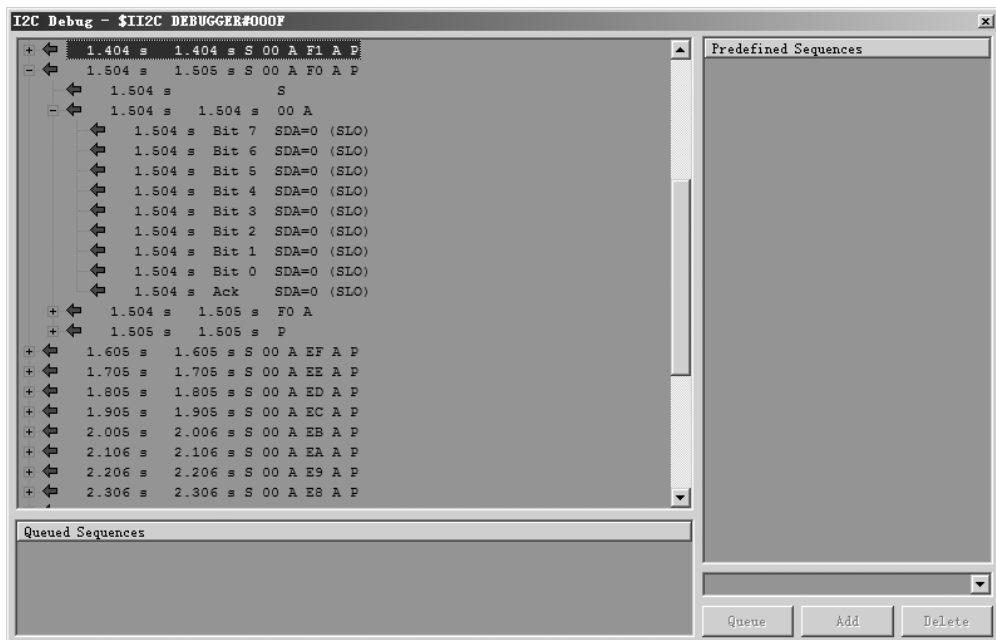


图 8.12 I<sup>2</sup>C 总线调试器模型

图 8.13 I<sup>2</sup>C 总线调试器的属性设置对话框

- Clock frequency in Hz: SCL 时钟频率。
- Address byte 1: 地址字节 1, 如果使用此终端仿真一个从元件, 则这一属性指定从器件的第一个地址字节。
- Address byte 2: 地址字节 2, 如果使用此终端仿真一个从元件, 并期望使用 10 位地址, 则这一属性指定从器件的第二个地址字节。

在实例的电路中添加一个 I<sup>2</sup>C 总线调试器, 并且分别连接其对应引脚, 点击运行, 可以看到如图 8.14 所示的总线传输数据。

图 8.14 I<sup>2</sup>C 总线调试器的运行窗口



### 8.9.2 ATmega16 双机使用 SPI 总线模块进行通信

本应用是两片 ATmega16 使用 SPI 总线接口进行通信的实例，单片机 A（U1）通过 SPI 总线接口将数据发送到单片机 B（U2），单片机 B 接收到数据之后将其从串口送出。

#### 1. 实例的设计思路

实例的设计思路可以参考 8.7 节。

#### 2. 实例的 Proteus 电路

实例的 Proteus 电路如图 8.15 所示，U1 作为主机，使用 SPI 接口总线和作为从机的 U2 通信，在 U2 的 TXD 引脚上添加了一个虚拟终端用于监视串口的输出，实例涉及的 Proteus 器件如表 8.20 所示。

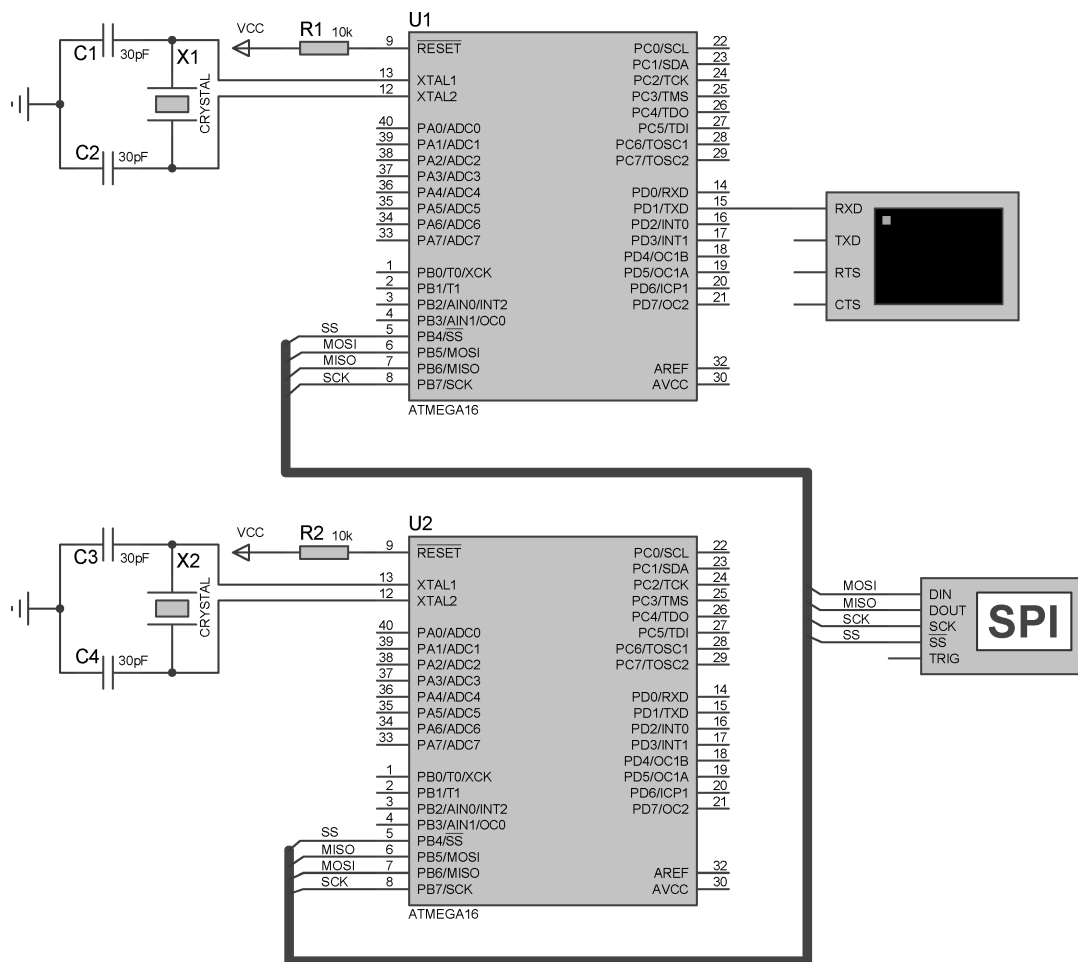


图 8.15 实例的 Proteus 电路



表 8.20 应用实例器件列表

器件名称	大 类 库	子 类 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体

### 3. 实例的应用代码

实例的应用代码如例 8.3 和 8.4 所示。

代码首先初始化 SPI 总线模块，然后调用 SPI\_Transmit 函数来发送数据，在每次发送数据之间有 100ms 的延时。

#### 【例 8.3】 SPI 总线双机通信主机代码

```
#include <iom16v.h>
#include <macros.h>

//延时相关函数
void Delayus(unsigned intUS)
{
    unsigned int i;
    US = US * 5/4;          //5/4 是在 8MHz 晶振下，通过软件仿真反复实验得到的数值
    for(i=0;i<US;i++);
}
void Delayms(unsigned int MS)
{
    unsigned int i,j;
    for(i=0;i<MS;i++)
        for(j=0;j<1141;j++); //1141 是在 8MHz 晶振下，通过软件仿真反复实验得到的数值
}
//端口初始化
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
```



```

}
//SPI 总线初始化
void spi_init( void)
{
    SPCR = 0x70;           //fosc/8
    SPSR = 0x00;           //2 倍
    DDRB = 0XB0;
    PORTB |= 0X40;
}
//ATmega16 初始化代码
void init_devices( void)
{
    CLI();
    port_init();
    spi_init();
    SEI();
}
//SPI 总线数据发送函数
void SPI_Transmit( char cData)
{
    //启动数据传输
    SPDR = cData;
    //等待传输结束
    while( ! ( SPSR & ( 1 << SPIF) ) );
}
//SPI 总线数据接收函数
unsigned char SPI_Receive( void)
{
    //等待接收结束
    while( ! ( SPSR & ( 1 << SPIF) ) );
    //返回数据
    return SPDR;
}
void main( void)
{
    unsigned char i;
    init_devices();
    Delayms( 1 );           //延时 1ms
    while( 1 )
    {
        for( i = 255; i > 0; i -- )    //循环发送

```



```
{  
    SPI_Transmit(i);  
    Delayms(100);          //延时  
}  
  
}
```

从机使用 SPI 中断来进行接收, 当接收到数据之后触发 SPI 的中断事件, 在中断服务子程序中调用 SPI\_Receive 来接收一个字节的数据, 然后把接收到的数据调用 putchar 函数通过 USART 送出。

#### 【例 8.4】 SPI 总线双机通信从机代码

```
#include <iom16v.h>  
#include <macros.h>  
unsigned char temp;          //用于存放接收到的数据  
//putchar 串口发送函数  
extern void putchar(char inputdata)  
{  
    while(! (UCSRA & (1 << UDRE)));  
    UDR = inputdata;  
}  
  
//端口初始化  
void port_init(void)  
{  
    PORTA = 0x00;  
    DDRA  = 0x00;  
    PORTB = 0x00;  
    DDRB  = 0x00;  
    PORTC = 0x00;  
    DDRC  = 0x00;  
    PORTD = 0x00;  
    DDRD  = 0x00;  
}  
  
//SPI 数据发送函数  
void SPI_Transmit(char cData)  
{  
    //启动数据传输  
    SPDR = cData;  
    //等待传输结束  
    while(! (SPSR & (1 << SPIF)));
```



```
}  
//SPI 数据接收函数  
unsigned char SPI_Receive( void)  
{  
    //等待接收结束  
    while( ! (SPSR & (1 << SPIF)) );  
    //返回数据  
    return SPDR;  
}  
//SPI 初始化函数  
void spi_init( void)  
{  
    SPCR = 0xE0;           //从机工作模式  
    SPSR = 0x00;           //清除  
}  
//SPI 中断服务子程序  
#pragma interrupt_handler spi_stc_isr:iv_SPI_STC  
void spi_stc_isr( void)  
{  
    //接收数据, 存放在 temp 中  
    temp = SPI_Receive();  
    putchar( temp);        //通过串口发送  
}  
//UART 初始化函数, 9600bps  
void uart0_init( void)  
{  
    UCSRB = 0x00;  
    UCSRA = 0x00;  
    UCSRC = BIT( URSEL ) | 0x06;  
    UBRRL = 0x33;  
    UBRRH = 0x00;  
    UCSRB = 0x48;  
}  
  
//初始化 ATmega16  
void init_devices( void)  
{  
    CLI();  
    port_init();  
    spi_init();  
    uart0_init();  
    SEI();  
}
```

```

}

void main(void)
{
    unsigned char temp = 0x01;
    init_devices();
    while(1)
    {
        // ...
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，可以在单片机 B 的 TXD0 引脚的虚拟终端上看到对应的数据输出，如图 8.16 所示。

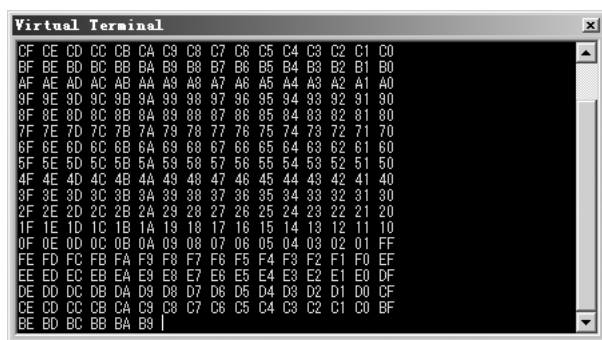


图 8.16 实例的仿真运行结果

### 总结

读者可以在单片机 A 上添加对应的串行模块驱动代码，使得单片机 A 可以从 PC 接收数据，然后通过 SPI 总线发送出去，这样就构成了一个 UART - SPI 总线桥。

#### 5. Proteus 中的 SPI DEBUGGER

SPI 总线调试器（SPI DEBUGGER）是用来观察当前 SPI 总线数据的虚拟仪器，在 Proteus 中，单击工具箱中的“Virtual Instrument Mode”按钮，此时当前窗口会出现包括 SPI 总线调试器的所有虚拟仪器的列表，在该列表中选择 SPI 总线调试器后，在电路图中点击即可放置，如图 8.17 所示，其引脚说明如下。

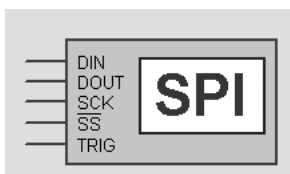


图 8.17 SPI 总线调试器模型

- DIN：串行数据接收引脚。
  - DOUT：串行数据输出引脚。
  - SCK：SPI 串行时钟引脚。
  - $\overline{SS}$ ：从模式选择端引脚，在从模式下该引脚必须为低电平才能使终端响应；主模式时当数据正传输时此端为低电平。
  - TRIG：输入引脚，能够把下一个存储序列放到 SPI 的输出序列中。
- 双击 SPI 总线调试器，弹出如图 8.18 的属性设置对话框，其中各个主要参数说明如下。
- SPI Mode：用于设置 SPI 总线调试器的工作方式，其有三种工作模式可选择，即 Monitor（监控模式）、Master（主模式）和 Slave（从模式）。
  - Master clock frequency in Hz：主模式的时钟频率，单位为 Hz。
  - SCK Idle state is：SCK 时钟空闲状态电平选择，要么为高（High），要么为低（Low）。
  - Sampling edge：采样边，指定 DIN 引脚采样的时钟边沿，选择 SCK 从空闲到激活（Idle to active）状态，或从激活到空闲（active to Idle）状态。
  - Bit order：位顺序，指定一个传输数据的位顺序，可先传送最高位 MSB，也可先传送最低位 LSB。

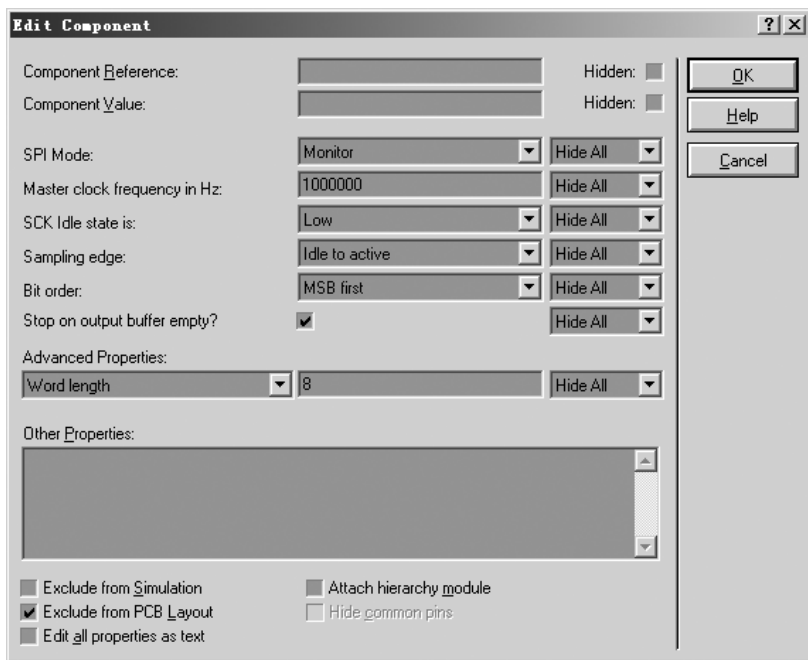
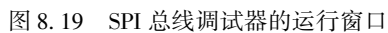


图 8.18 SPI 总线调试器的属性设置对话框

在实例的电路中添加一个 I<sup>2</sup>C 总线调试器并且分别连接其对应引脚，点击运行，可以看到如图 8.19 所示的总线传输数据。





# 第9章 ATmega16 单片机的比较器和 ADC 模块

ATmega16 单片机内置了一个模拟比较器和一个 ADC 转换模块，可用于实现模拟信号的输入和检测。

## 9.1 ATmega16 单片机的比较器

ATmega16 的比较器可以用作模拟电路和数字电路的接口，还可以用作波形产生和变换电路等，如利用简单电压比较器可将正弦波变为同频率的方波或矩形波。

### 9.1.1 模拟比较器基础

ATmega16 内置的模拟比较器可以看作放大倍数接近“无穷大”的运算放大器，可以用来比较两个电压的大小，然后得到一个逻辑“1”（高电平）或者逻辑“0”（低电平）的结果。

- 当正端输入端电压高于负端输入端电压时，比较器输出为高电平。
- 当正端输入端电压低于负端输入端电压时，比较器输出为低电平。

内部比较器对正输入引脚（AIN0，PB2）上的电压值与负输入引脚（AIN1，PB3）上的电压值进行比较，当 AIN0 引脚上的电压比负极 AIN1 引脚上的电压高时，模拟比较器的输出 ACO 被置“1”。该比较器的输出可用来触发定时器/计数器 1 的输入捕捉功能。此外，该比较器还可触发自己专有的、独立的中断，用户可以选择比较器是以上升沿、下降沿还是交替变化的边沿来触发中断。

### 9.1.2 模拟比较器的寄存器

ATmega16 对模拟比较器模块的控制也是通过对寄存器的操作完成的，其相关的寄存器包括模拟比较器的控制和状态寄存器 ACSR 和特殊功能 I/O 寄存器 SFIOR。

#### 1. 控制和状态寄存器 ACSR

模拟比较器的控制和状态寄存器 ACSR 用于控制模拟比较器的操作并且反应模拟比较器的状态，其内部结构如表 9.1 所示。

表 9.1 模拟比较器的控制和状态寄存器

BIT	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	R/W
0	0	N/A	0	0	0	0	0	0

- ACD：模拟比较器禁用标志位。当 ACD 被置“1”时，模拟比较器的电源被切断，可



可以在任何时候通过向此位写“1”来关掉模拟比较器。这可以减少器件工作模式及空闲模式下的功耗。在修改 ACD 位时，必须清零 ACSR 寄存器的 ACIE 位来禁止模拟比较器中断，否则 ACD 改变时可能会产生中断。

- **ACBG**：选择模拟比较器的参考源。当 ACBG 被置“1”时，模拟比较器的正极输入由参考源所取代，否则模拟比较器的正极输入为 AIN0 引脚上外加的电压。
- **ACO**：模拟比较器输出位。模拟比较器的输出经过同步后经过 1 ~ 2 个时钟周期的延时之后在 ACO 位上反映出来，如果  $AIN0 > AIN1$ ， $ACO = 1$ ，否则  $ACO = 0$ 。
- **ACI**：模拟比较器中断标志位。当比较器的输出事件触发了由 ACIS1 及 ACIS0 定义的中断模式时，ACI 位被置“1”。如果 ACIE 和 SREG 寄存器的全局中断标志 I 也被置“1”，则触发模拟比较器中断事件，同时 ACI 位被硬件清零。ACI 也可以通过向该位写入“1”来清除。
- **ACIE**：模拟比较器中断使能位。当 ACIE 位被置“1”且状态寄存器中的全局中断标志 I 也被置“1”时，模拟比较器中断被激活，否则中断被禁止。
- **ACIC**：模拟比较器输入捕捉使能位。在 ACIC 置位后允许通过模拟比较器来触发 T/C1 的输入捕捉功能。此时比较器的输出被直接连接到输入捕捉的前端逻辑，从而使得比较器可以利用 T/C1 输入捕捉中断逻辑的噪声抑制器及触发沿选择功能。ACIC 为“0”时，模拟比较器及输入捕捉功能之间没有任何联系。为了使比较器可以触发 T/C1 的输入捕捉中断，定时器中断屏蔽寄存器 TIMSK 的 TICIE1 也必须置位。
- **ACIS1、ACIS0**：模拟比较器中断模式选择位。这两位确定触发模拟比较器中断的事件如表 9.2 所示，如果需要改变 ACIS1 和 ACIS0，则必须先清零 ACSR 寄存器的中断使能位来禁止模拟比较器中断，否则有可能在改变这两位时产生中断事件。

表 9.2 ACIS1 和 ACIS0 的设置

ACIS1	ACIS0	中断模式
0	0	比较器输出发生变化时触发中断
0	1	保留
1	0	比较器输出下降沿触发中断
1	1	比较器输出上升沿触发中断

## 2. 特殊功能 I/O 寄存器 SFIOR

SFIOR 寄存器中的模拟比较器多路复用器使能位 ACME，当此位被置“1”且 ATmega16 的 ADC 模块处于关闭状态（ADCSRA 寄存器的 ADEN 位为“0”）时，ADC 多路复用器为模拟比较器选择负极输入，当此位为“0”时，比较器的负极输入端使用 AIN1 的输入。

表 9.3 是 SFIOR 寄存器的内部位结构。

表 9.3 特殊功能 I/O 寄存器 SFIOR

BIT	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSR2	PSR10
R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
0	0	0	0	0	0	0	0	0



### 9.1.3 模拟比较器的输入通道

除了 AIN0 (PB2) 和 AIN1 (PB3) 引脚之外, 可以选择 ADC0 ~ ADC7 (PA0 ~ PA7) 引脚中的的任意一个引脚来代替模拟比较器的负极输入端, 并且使用 ADC 复用器来完成切换。为了使用这个功能必须先关掉 ADC, 如果模拟比较器复用器使能位 (SFIOR 中的 ACME) 被置位, 且 ADC 也已经关掉 (ADCSRA 寄存器的 ADEN 为 0), 则可以通过 ADMUX 寄存器的 MUX2 ~ MUX0 来选择替代模拟比较器负极输入的引脚, 如表 9.4 所示。如果 ACME 位被清零或 ADEN 被置位, 则模拟比较器的负极输入为 AIN1 引脚上加的电压。

表 9.4 模拟比较器复用输入

ACME	ADEN	MUX2 ~ MUX0	模拟比较器负输入
0	X	XXX	AIN1
1	1	XXX	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6
1	0	111	ADC7

## 9.2 ATmega16 单片机的 ADC 模块

ADC (或 A/D, Analog to Digital Converter) 芯片是将模拟信号转换成数字信号的器件, 其原理是将时间连续变化的模拟量转换为离散的量化数值, 整个过程通常经历采样、量化和编码三个步骤, 如图 9.1 所示。

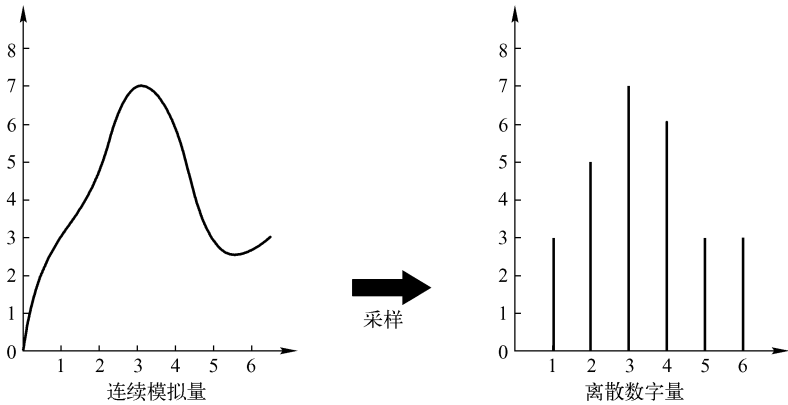


图 9.1 模拟数字转换



ATmega16 的内置 ADC 模块是逐次逼近型，它可以把外部的模拟信号转化为数字信号，具有以下的特点。

- 具有 10 位采样精度，提供 0.5LSB 的非线性度和  $\pm 2$  LSB 的绝对精度。
- 采样转化时间为  $65 \sim 260\mu\text{s}$ ，当使用最高分辨率时采样率高达 15kSPS。
- 8 路复用的单端输入通道和 8 路差分输入通道，并且有 2 路可选增益为  $10 \times$  与  $200 \times$  的差分输入通道。
- ADC 转换结果可以选择左对齐。
- 支持 0V 至  $V_{\text{cc}}$  电压值的输入电压范围，并且可选 2.56V 作为 ADC 参考电压。
- 有连续转换或单次转换模式，并且支持自动触发中断源启动转换。

### 9.2.1 ADC 模块基础

ATmega16 的 ADC 模块与一个 8 通道的模拟多路复用器连接，能对来自 PA0 ~ PA7 的 8 路单端输入电压进行采样，该单端电压输入以 0V (GND) 为基准。同时，ATmega16 还支持 16 路差分电压输入组合。两路差分输入 (ADC1、ADC0 和 ADC3、ADC2) 内部有可编程增益级，在 ADC 转换前可以给差分输入电压提供 0dB ( $1 \times$ )、20dB ( $10 \times$ ) 或 46dB ( $200 \times$ ) 的放大级。七路差分模拟输入通道共享一个通用负端 (ADC1)，而其他任何 ADC 输入均可作为正输入端。如果使用  $1 \times$  或  $10 \times$  增益，可得到 8 位分辨率，如果使用  $200 \times$  增益，可以得到 7 位分辨率。

ATmega16 的 ADC 模块内置了一个采样保持电路，能确保在转换过程中输入到 ADC 的电压保持恒定。

ATmega16 的 ADC 模块由 AVCC 引脚单独提供电源，AVCC 与  $V_{\text{cc}}$  之间的偏差不能超过  $\pm 0.3\text{V}$ ，它还内置了一个标称值为 2.56V 的基准电压，可以通过在 AREF 引脚上加一个电容进行解耦，以更好地抑制噪声。

ATmega16 的 ADC 模块通过逐次逼近的方法将输入的模拟电压转换成一个 10 位的数字量，其最小值（所有数值位为 0）代表 GND，最大值（所有数值位为 1）代表 AREF 引脚上的电压再减去 1 LSB。

通过对 ADMUX 寄存器 REFSn 位的写操作，可以把 AVCC 或内部 2.56V 的参考电压连接到 AREF 引脚作为 ADC 模块的参考电压，在 AREF 引脚上外加电容可以对片内参考电压进行解耦以提高噪声抑制性能。

模拟输入通道与差分增益可以通过写 ADMUX 寄存器的 MUX 位来选择。任何 ADC 输入引脚，如 GND 以及固定能隙参考电压，都可以作为 ADC 的单端输入，同时 ADC 输入引脚可选做差分增益放大器的正或负输入。如果选择差分通道，通过选择被选输入信号对的增益因子可以得到电压差分放大级，然后被放大之后的电压将作为 ADC 的模拟输入；如果使用单端通道，将无视增益放大器。通过设置 ADCSRA 寄存器的 ADEN 位即可启动 ADC，只有当 ADEN 置位时，参考电压及输入通道选择才生效，当 ADEN 被清零时，ADC 并不耗电，因此建议在进入节能睡眠模式之前关闭 ADC。

ADC 转换结果为 10 位，存放于 ADC 数据寄存器 ADCH 及 ADCL 中。默认情况下转换结果为右对齐，但可通过设置 ADMUX 寄存器的 ADLAR 位修改为左对齐。如果要求转换结果



左对齐，且最高只需 8 位的转换精度，那么只需要读取 ADCH 寄存器内的数据，否则要先读 ADCL 寄存器，再读 ADCH 寄存器，以保证数据寄存器中的内容是同一次转换的结果。一旦对 ADCL 寄存器进行读操作，ADC 模块对数据寄存器的寻址就被阻止了，也就是说，读取 ADCL 寄存器之后，即使在读 ADCH 寄存器之前又有一次 ADC 转换结束，数据寄存器的数据也不会更新，从而保证了转换结果不丢失。当 ADCH 寄存器内的数据被读出后，ADC 即可再次访问 ADCH 及 ADCL 寄存器。

ADC 转换结束可以触发中断，即使由于转换发生在读取 ADCH 寄存器与 ADCL 寄存器之间而造成 ADC 无法访问数据寄存器，并因此丢失了转换数据，仍会触发中断事件。

## 9.2.2 ADC 模块的寄存器

ATmega16 同样通过对寄存器的操作实现对 ADC 模块的控制，这些相关的寄存器包括 ADC 多工选择寄存器 (ADMUX)、ADC 控制和状态寄存器 (ADCSRA)，ADC 数据寄存器 (ADCL 和 ADCH) 和 ADC 特殊功能 I/O 寄存器 (SFIOR)。

### 1. ADC 多工选择寄存器 ADMUX

ADC 多工选择寄存器 ADMUX 主要用于控制 ADC 模块的参考电压、转化结果数据对齐方式以及增益倍数设置，表 9.5 是 ADMUX 的内部位结构。

表 9.5 ADC 多工选择寄存器 ADMUX

BIT	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- REFS1 和 REFS0：参考电压选择位，其使用方法参考表 9.6，需要注意的是，如果在 ADC 转换过程中改变了 REFS1 和 REFS0 的设置，只有等到当前转换结束 (ADCSRA 寄存器的 ADIF 置位) 之后，该改变才会起作用，并且如果在 AREF 引脚上施加了外部参考电压，就不能使用内部参考电压。

表 9.6 ADC 参考电压选择

REF1	REF0	参考电压选择
0	0	AREF，内部 VREF 关闭
0	1	AVCC，AREF 引脚外加滤波电容
1	0	保留
1	1	2.56V 的片内基准电压源，AREF 引脚外接滤波电容

- ADLAR：ADC 转换结果左对齐位。ADLAR 位置用于设置 ADC 的转换结果在 ADC 数据寄存器中的存放形式，当 ADLAR 被置“1”时，ADC 转换结果为左对齐，否则为右对齐。ADLAR 的改变将立即影响 ADC 数据寄存器的内容，不论是否有转换正在进行。
- MUX4 ~ MUX0：模拟通道与增益选择位，用于选择连接到 ADC 引脚的模拟输入以及控制查封通道的增益，如表 9.7 所示。



表 9.7 输入通道和增益选择

MUX4 ~ MUX0	单 端 输 入	正差分输入	负差分输入	增 益
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC8			
01000	N/A	ADC0	ADC0	10 ×
01001		ADC1	ADC0	10 ×
01010		ADC0	ADC0	200 ×
01011		ADC1	ADC0	200 ×
01100		ADC2	ADC2	10 ×
01101		ADC3	ADC2	10 ×
01110		ADC2	ADC2	200 ×
01111		ADC3	ADC2	200 ×
10000		ADC1	ADC1	1 ×
10001		ADC2	ADC1	1 ×
10010		ADC3	ADC1	1 ×
10100		ADC4	ADC1	1 ×
10101		ADC5	ADC1	1 ×
10110		ADC6	ADC1	1 ×
10111		ADC8	ADC1	1 ×
11000		ADC0	ADC2	1 ×
11001		ADC1	ADC2	1 ×
11010		ADC2	ADC2	1 ×
11011		ADC3	ADC2	1 ×
11100	N/A	ADC4	ADC2	1 ×
11101		ADC5	ADC2	1 ×
11110	1.22V	N/A		
11111	0V			



## 注意

如果在 ADC 转换过程中改变了 MUX4 ~ MUX0 的设置，只有等到当前转换结束（ADCSRA 寄存器的 ADIF 置位）之后，该改变才会起作用。



## 2. ADC 控制和状态寄存器 ADCSRA

ADC 控制和状态寄存器 ADCSRA 用于对 ADC 的使能控制以及返回对应的状态，其内部位结构如表 9.8 所示。

表 9.8 ADC 控制和状态寄存器 ADCSRA

BIT	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- ADEN: ADC 使能控制位。当 ADEN 置“1”时启动 ADC，否则关闭 ADC 功能，如果在转换过程中对 ADEN 清零，将关闭 ADC，立即中止正在进行的转换。
- ADSC: ADC 启动转换控制位。在单次转换工作模式下，ADSC 置“1”将启动一次 ADC 转换；在连续转换工作模式下，ADSC 置位将启动首次转换。第一次转换（ADC 启动之后置位 ADSC，或者在使能 ADC 的同时置位 ADSC）需要 25 个 ADC 时钟周期，而不是正常情况下的 13 个。第一次转换执行 ADC 初始化的工作。在转换进行过程中读取 ADSC 的返回值为“1”，直至转换结束。ADSC 的清零不产生任何动作。
- ADATE: ADC 自动触发使能位。ADATE 的置位将启动 ADC 自动触发功能，触发信号的上升沿启动 ADC 转换，在这种工作状态下，触发信号源通过对 SFIOR 寄存器的 ADC 触发信号源选择位 ADTS 的设置来选择。
- ADIF: ADC 中断标志位。当 ADC 转换结束且数据寄存器被更新后，ADIF 位被置位。如果 ADIE 及 SREG 中的全局中断使能位 I 也被置位，将产生一个 ADC 转换结束中断事件。当进入 ADC 中断服务程序时，ADIF 被硬件自动清零，也可以通过向此 ADIF 位写入“1”来清除。



### 注意

如果对 ADCSRA 位进行读 - 修改 - 写操作，那么待处理的中断会被禁止。

- ADIE: ADC 中断使能位。当 ADIE 位以及 SREG 的位 I 置“1”，则使能 ADC 转换结束中断。
- ADPS2、ADPS1 和 ADPS0: ADC 预分频选择位，用于确定系统工作时钟和 ADC 输入时钟之间的分频因子，其对应关系如表 9.9 所示。

表 9.9 ADC 分频选择

ADPS2	ADPS1	ADPS0	分频因子
0	0	0	2
0	0	1	2
0	1	0	4
1	0	0	8
1	0	1	16
1	1	0	32





续表

ADPS2	ADPS1	ADPS0	分频因子
1	1	0	64
1	1	1	128

### 3. ADC 数据寄存器 ADCL 和 ADCH

ADC 数据寄存器 ADCL 和 ADCH 用于存放 ADC 的转换结果，其内部结构如表 9.10 和表 9.11 所示，需要注意的是，在左端对齐和右端对齐时，这两个寄存器有不同的内部结构。

表 9.10 ADCL 和 ADCH (ADLAR = 0)

ADCH	—	—	—	—	—	—	ADC9	ADC8
ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
读/写	R	R	R	R	R	R	R	R
读/写	R	R	R	R	R	R	R	R
初始值	0	0	0	0	0	0	0	0
初始值	0	0	0	0	0	0	0	0

表 9.11 ADCL 和 ADCH (ADLAR = 1)

ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
ADCL	ADC1	ADC0	—	—	—	—	—	—
读/写	R	R	R	R	R	R	R	R
读/写	R	R	R	R	R	R	R	R
初始值	0	0	0	0	0	0	0	0
初始值	0	0	0	0	0	0	0	0

当 ADC 转换结束后，其转换结果将保存在这两个寄存器中，如果采用差分通道，其结果将由 2 的补码形式表示。

读取 ADCL 寄存器之后，ADC 的数据寄存器一直要等到 ADCH 寄存器内的数据也被读出才可以进行数据更新。因此，如果转换结果为左对齐，且要求的精度不高于 8bit，那么仅需要读取 ADCH，否则必须先读出 ADCL 再读 ADCH。

ADMUX 寄存器的 ADLAR 位以及 MUXn 位会影响转换结果在数据寄存器中的表示方式，如果 ADLAR 位置“1”，那么结果为左对齐，反之则为右对齐（系统默认方式）。

### 4. ADC 特殊功能 I/O 寄存器 SFIOR

ADC 特殊功能 I/O 寄存器 SFIOR 的内部结构如表 9.12 所示，其中只有最高的 3 位影响 ADC 模块的工作，用于选择自动 ADC 触发源。

表 9.12 ADC 特殊功能 I/O 寄存器 SFIOR

BIT	ADTS2	ADTS1	ADTS0	—	ACME	PUD	PSAR2	PSR10
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0



ADTS2 ~ ADTS0: ADC 自动触发源选择。若 ADCSRA 寄存器的 ADATE 位置“1”, 则由 ADTS 的值来确定触发 ADC 转换的触发源, 否则 ADTS 的设置被忽视。被选中自动触发源 (中断标志) 在其上升沿触发 ADC 转换, 从一个中断标志清零的触发源切换到中断标志置位的触发源会使触发信号产生一个上升沿。如果此时 ADCSRA 寄存器的 ADEN 位为“1”, 则立即启动 ADC 转换。如果切换到连续运行模式 (ADTS[2:0] = 0) 之后, 即使 ADC 中断标志已经置位也不会产生触发事件, ADC 触发源的选择如表 9.13 所示。

表 9.13 ADC 触发源的选择

ADTS2	ADTS1	ADTS0	触 发 源
0	0	0	连续转换模式
0	0	1	模拟比较器
0	1	0	外部中断 0
1	0	0	定时器/计数器 0 比较匹配
1	0	1	定时器/计数器 1 溢出
1	1	0	定时器/计数器 0 比较匹配
1	1	1	定时器/计数器 1 溢出



## 说明

自动源触发的工作过程是当有被选择的中断事件产生时, 立即启动一次 ADC 转换。

### 9.2.3 ADC 模块的转换过程

#### 1. ADC 转换启动

向 ATmega16 的启动转换位 ADSC 位写入“1”可以启动单次转换, 在转换过程中此位应该保持为高, 一直到转换结束, 然后被硬件清零。如果在转换过程中选择了另一个通道, 那么 ADC 会完成当前通道的转换, 然后改变通道。

ADC 模块的转换支持不同的触发源, 也就是说, 可以用不同的事件来触发一次 ADC 转换, 当 ADCSRA 寄存器的 ADC 自动触发允许位 ADATE 被使能时, ADCSRB 寄存器的 ADC 触发选择位 ADTS 选择的触发源将自动触发转换。

当所选的触发信号产生上升沿时, ADC 预分频器复位并开始转换。这提供了一个在固定时间间隔下启动转换的方法。转换结束后即使触发信号仍然存在, 也不会启动一次新的转换。如果在转换过程中触发信号中又产生了一个上升沿, 这个上升沿将被忽略。即使特定的中断被禁止或全局中断使能位为“0”, 中断标志仍将置位。这样可以在不产生中断的情况下触发一次转换。但是为了在下次中断事件发生时触发新的转换, 必须将中断标志清零。

如果使用 ADC 的中断标志作为触发源, 则可以在正在进行的转换结束后立即开始下一次转换, 这样 ADC 就进入连续转换模式, 持续地进行采样、转换并且刷新对 ADC 数据寄存器。在这种工作模式下, 后续的 ADC 转换不依赖于 ADC 中断标志 ADIF 是否置位。



## 注意

第一次转换需要通过向 ADCSRA 寄存器的 ADSC 位写“1”来启动。



如果使能了自动触发，置位 ADCSRA 寄存器的 ADSC 将启动单次转换。ADSC 标志位还可用来检测转换是否在进行之中。不论转换是如何启动的，在转换进行过程中 ADSC 将一直为“1”。

## 2. ADC 转换的预分频以及时序

由于采取了逐次逼近的采样原理，ADC 模块的逐次逼近电路需要一个从 50 ~ 200kHz 的输入时钟作为驱动以获得最大精度，如果所需的转换精度低于 10bit，那么输入时钟频率可以高于 200kHz，以达到更高的采样率。

ADC 模块中有一个预分频器，它可以将超过 100kHz 的系统工作时钟分频产生需要的各种 ADC 时钟，这个预分频器通过 ADCSRA 寄存器的 ADPS 位进行设置，置位 ADCSRA 寄存器的 ADEN 位置将使能 ADC，预分频器开始计数，并且持续到 ADEN 被清零。

ADCSRA 寄存器的 ADSC 位置“1”后，单端转换在下一个 ADC 时钟周期的上升沿开始启动，正常转换需要 13 个 ADC 时钟周期。



### 注意

为了初始化模拟电路，ADC 使能（ADCSRA 寄存器 ADEN 位被置位）后的第一次转换需要 25 个 ADC 时钟周期。

在普通的 ADC 转换过程中，采样保持在转换启动之后的 1.5 个 ADC 时钟到来时开始；而第一次 ADC 转换的采样保持则发生在转换启动之后的 13.5 个 ADC 时钟。转换结束后，ADC 结果被送入 ADC 数据寄存器，且 ADIF 标志置位。ADSC 同时清零（单次转换模式）。之后软件可以再次置位 ADSC 标志，从而在 ADC 时钟的第一个上升沿启动一次新的转换。使用自动触发时，触发事件发生将复位预分频器，这保证了触发事件和转换启动之间的延时是固定的。在此自动触发的工作模式下，采样保持在触发信号上升沿之后的 2 个 ADC 时钟发生，为了实现同步逻辑则需要额外的 3 个 CPU 时钟周期。

如果使用差分输入模式，每次转换要 25 个 ADC 时钟周期，这是由于每次转换后 ADC 必须禁用再重新使能，自动触发源为 ADC 转换结束外除外。

在连续转换模式下，当 ADSC 置“1”时，只要上一次转换一结束立即开始下一次转换，ADC 的转换时间如表 9.14 所示。

表 9.14 ADC 的转换时间

工作模式	采样 & 保持时钟周期数	转换周期
第一次转换	14.5	25
单端正常转换	1.5	12
自动出发转换	2	13.5
正常转换/差分	1.5/2.5	13/14

## 3. ADC 模块的差分增益通道

当 ADC 使用差分增益通道时，需要考虑转换的确定特征。在进行差分转换时，ADC 模块与内部时钟  $CK_{ADC2}$  同步，该时钟的频率为 ADC 时钟频率的 1/2，其同步是由 ADC 模块在



$CK_{ADC2}$  边沿出现采样与保持时自动实现的。当  $CK_{ADC2}$  为低时, 通过启动转换 (包括单次转换与第一次连续转换) 将与单端转换使用的时间相同, 即预分频后的 13 个 ADC 时钟周期。当  $CK_{ADC2}$  为高时, 由于同步机制, 将会使用 14 个 ADC 时钟周期。在连续转换工作模式下, 一次 ADC 转换结束后立即启动新的转换, 由于  $CK_{ADC2}$  此时为高, 除第一次外的所有自动启动将使用 14 个 ADC 时钟周期。

在所有的增益设置中, 当带宽为 4kHz 时增益级最优, 更高的频率可能会造成非线性放大。当输入信号包含高于增益级带宽的频率时, 应在输入前加入低通滤波器。需要注意的是, ADC 时钟频率不受增益级带宽限制, 例如, 不管通道带宽是多少, ADC 的时钟周期均为  $6\mu s$ , 允许通道采样率为 12 kSPS。

如果使用差分增益通道且通过自动触发启动转换, 在转换时 ADC 必须关闭, 当使用自动触发时, ADC 预分频器在转换启动前复位。由于在转换前的增益级依靠稳定的 ADC 时钟, 该转换无效。在每次转换 (对寄存器 ADCSRA 的 ADEN 位中清零后立刻置“1”), 通过禁用, 然后重使能 ADC, 只执行扩展转换, 扩展转换结果有效。

#### 9.2.4 ADC 模块的输入通道和参考电源

所有 ADC 模块都需要一个作为基准源的参考电压, 由于 ATmega16 的 ADC 模块拥有多个输入通道, 所以在使用 ADC 模块时需要选择合适的参考电源和输入通道。

ADC 模块的 ADMUX 寄存器中的  $MUX_n$  及 REFS1 和 REFS0 位通过临时寄存器实现了单缓冲, ATmega16 可对此临时寄存器进行随机访问, 以保证在转换过程中通道和基准源的切换发生于安全的时刻。在转换启动之前, 通道及基准源的选择可随时进行。一旦转换开始就不允许再选择通道和基准源了, 从而保证 ADC 有充足的采样时间。在转换完成 (ADCSRA 寄存器的 ADIF 被置位) 之前的最后一个时钟周期, 通道和基准源的选择又可以重新开始。转换的开始时刻为 ADSC 置位后的下一个时钟的上升沿。



##### 说明

建议用户在置位 ADSC 之后的一个 ADC 时钟周期里, 不要操作 ADMUX 以选择新的通道及基准源。

使用自动触发时, 触发事件发生的时间是不确定的, 为了控制新设置对转换的影响, 在更新 ADMUX 寄存器时一定要特别小心。若 ADATE 及 ADEN 位都置“1”, 则中断事件可以在任意时刻发生, 如果在此期间改变 ADMUX 寄存器的内容, 用户就无法判别下一次转换是基于旧的设置还是最新的设置, 推荐在以下三种时刻对 ADMUX 进行更新。

- (1) ADATE 位或者 ADEN 位为 0。
- (2) 在转换过程中, 但在触发事件发生后至少一个 ADC 时钟周期。
- (3) 转换结束之后, 但在作为触发源的中断标志清零之前。

如果在上面提到的任意一种情况下更新 ADMUX, 那么新设置将在下一次 ADC 转换进行时生效。当改变差分通道时要特别注意: 一旦选定差分通道, 增益级要用  $125\mu s$  来稳定该值。因此, 在选定新通道后的  $125\mu s$  内不应该启动转换, 或舍弃该时间段内的转换结果。同样, 当通过改变 ADMUX 寄存器中的 REFS1 和 REFS0 位修改了 ADC 参考电源后的第一次转

换也要进行延时或者舍弃转换结果。

### 1. 选择 ADC 的输入通道

可以通过修改 ATmega16 的 ADMUX 寄存器的 MUX4 ~ MUX0 位来修改 ADC 的输入通道, 在进行选择的时候需要遵循以下原则。

(1) 当 ADC 工作于单次转换模式时, 总是在启动转换之前选定通道, 在 ADSC 被置位后的第一个 ADC 时钟周期就可以选择新的模拟输入通道了, 但是最简单的办法是等待转换结束后再改变通道。

(2) 当 ADC 工作于连续转换模式时, 总是在第一次转换开始之前选定通道, 在 ADSC 被置位后的第一个 ADC 时钟周期就可以选择新的模拟输入通道了, 但是最简单的办法是等待转换结束后再改变通道。但此时新一次转换已经自动开始了, 下一次的转换结果反映的是以前选定的模拟输入通道, 以后的转换结果才是新通道的。

(3) 当 ADC 切换到差分增益通道时, 由于自动偏移抵消电路需要沉积时间, 第一次转换结果准确率很低, 最好舍弃第一次转换结果。

### 2. 选择 ADC 的基准电压

ADC 的参考电压源 (VREF) 反映了 ADC 的转换范围, 若单端通道电平超过 VREF, 其结果将无限接近 0x3FF, 这个参考电源可以是 AVCC、内部 2.56V 基准或外加在 AREF 引脚上的电压。

AVCC 通过一个无源开关与 ADC 模块连接, 片内的 2.56V 参考电压由能隙基准源 (V<sub>BG</sub>) 通过内部放大器产生, 无论使用何种基准源, AREF 都直接与 ADC 模块相连, 通过在 AREF 与地之间外加电容可以提高参考电压的抗噪性。VREF 可以通过高输入内阻的伏特表在 AREF 引脚测得, 由于 VREF 的阻抗很高, 因此只能连接容性负载。

如果在 AREF 引脚上加上一个固定电压, ATmega16 则不能选择其他的基准源, 因为这会导致片内基准源与外部参考源的短路, 如果 AREF 引脚上没有连接任何外部参考源, 用户可以选择 AVCC 或 2.56V 作为基准源。



注意

参考源改变后的第一次 ADC 转换结果可能不准确, 建议舍弃这一次的转换结果。

## 9.2.5 ADC 模块的转换结果和精度定义

ADC 的转换结果存放于 ADCL 和 ADCH 寄存器中, 其中单次转换的结果见以下公式:

$$ADC = \frac{V_{in} \times 1024}{V_{REF}}$$

其中,  $V_{in}$  为待转换电压,  $V_{REF}$  为参考电压; 对于转换结果来说, 0x000 表示输入电压为 0V, 0x3FF 表示输入电压为参考电压的值减去 1LSB (最小刻度单位)。

如果使用差分通道, 其转换值则见以下公式:

$$ADC = \frac{(V_{POS} - V_{NEG}) \times GAIN \times 512}{V_{REF}}$$



其中,  $V_{POS}$  为输入引脚正电压;  $V_{NEG}$  为输入引脚负电压; GAIN 为使用的增益因子; 其转换结果用 2 的补码形式表示, 从 0x200 (−512d) 到 0x1FF (+511d), 如果需要快速判别电压的正负, 则可以读 ADCH 寄存器中的 ADC9 位, 如果为“1”, 则结果为负, 否则为正。

表 9.15 为 ADC 输入电压和输出码的对应关系, 例 9.1 为输出值和电压的对应计算方式。

表 9.15 ADC 输入电压和输出码的对应关系

$V_{ADCn}$	读 出 码	对应的十进制值
$V_{ADCn} + V_{REF}/GAIN$	0x1FF	511
$V_{ADCn} + 0.999V_{REF}/GAIN$	0x1FF	511
$V_{ADCn} + 0.998V_{REF}/GAIN$	0x1FE	510
$\vdots$	$\vdots$	$\vdots$
$V_{ADCn} + 0.001V_{REF}/GAIN$	0x001	1
$V_{ADCn}$	0x000	0
$V_{ADCn} - 0.001V_{REF}/GAIN$	0x3FF	−1
$\vdots$	$\vdots$	$\vdots$
$V_{ADCn} - 0.999V_{REF}/GAIN$	0x201	−511
$V_{ADCn} - V_{REF}/GAIN$	0x200	−512

### 【例 9.1】ADC 模块输出值的计算方法

ADMUX = 0xED; 对应 ADC3 − ADC2, 10 × 增益, 2.56V 参考电压, 左对齐方式。

当 ADC3 引脚上输入电压为 300mV, ADC2 引脚上电压为 500mV 时:

$$ADCR = \frac{512 \times 10 \times (300 - 500)}{2560} = -400 = 0x270$$

ADCL 的值为 0x00 且 ADCH 的值为 0x9C, 如果给 ADLAR 清零选择右对齐方式, 则 ADCL = 0x80, ADCH = 0x02。

ADC 模块的精度定义为一个  $n$  位的单端 ADC 将 GND 与 VREF 之间的线性电压转换成  $2n$  个 (LSBs) 不同的数字量, 其中最小的转换码为 0, 最大的转换码为  $2n - 1$ , 用以下的几个参数来定量描绘实际测量值和理论值的差异。

- 偏移: 起始 ADC 转换值 (0x000 到 0x001) 与理想转换 (0.5LSB) 之间的偏差, 其理想值为 0LSB。
- 增益误差: 在调整偏移之后, 最终 ADC 转换 (0x3FE 到 0x3FF) 与理想情况 (最大值以下 1.5LSB) 之间的偏差, 其理想值为 0LSB。
- 整体非线性 (INL): 在调整偏移及增益误差之后, 所有实际转换与理想转换之间的最大误差值, 其理想值为 0LSB。
- 差分非线性 (DNL): 实际码宽 (两个邻近转换之间的码间距) 与理论码宽 (1LSB) 之间的偏差, 其理论值为 0LSB。
- 量化误差: 由于输入的连续电压值被量化成有限位的数码, 在某个范围的输入电压 (1LSB) 被转换为相同的数码, 量化误差总是为  $\pm 0.5LSB$ 。



- 绝对精度：所有未经调整的实际转换值与理论转换值之间的最大偏差，由偏移、增益误差、差分误差、非线性及量化误差构成，其理想值为  $\pm 0.5\text{LSB}$ 。

### 9.3 ATmega16 比较器的应用实例

#### 9.3.1 双通道模拟信号比较应用实例

本应用是一个使用 ATmega16 的内置比较器进行电压比较的实例，当 AIN0 引脚上的电压不低于 AIN1 引脚时，数码管显示“0”，反之显示“1”。

##### 1. 实例的设计思路

ATmega16 对 AIN0 和 AIN1 上的电压进行比较，然后通过 PD 端口将比较结果对应的字形编码输出驱动数码管显示。

##### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 9.2 所示，使用 RV1 和 RV2 两个滑动变阻器对电压进行分压之后连接到 AIN0 和 AIN1 引脚作为电压输入，使用 PORTD 端口驱动了一个 7 段数码管。表 9.16 为实例电路使用的 Proteus 器件列表。

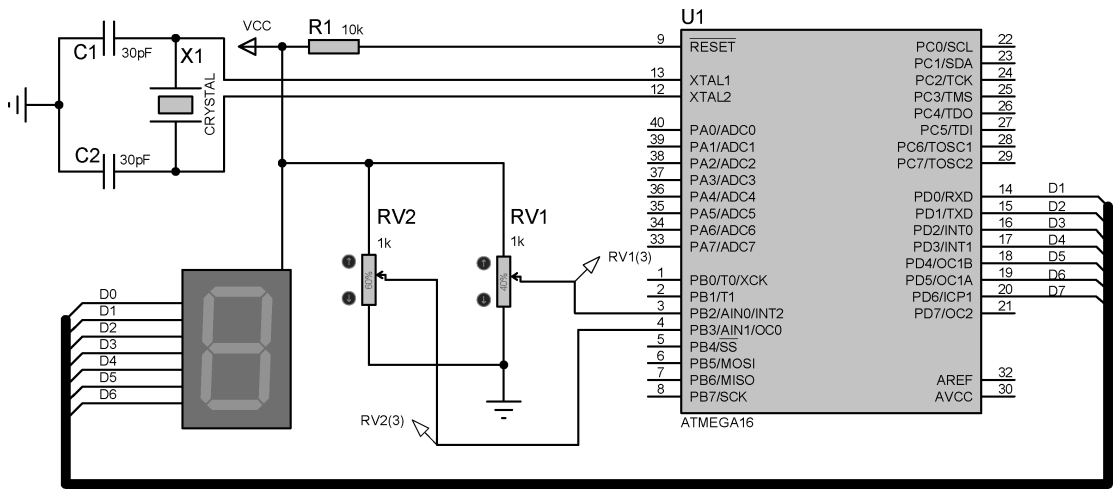


图 9.2 实例的 Proteus 电路

表 9.16 应用实例器件列表

器件名称	大类库	子类库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
7SEG - COM - ANODE	Optoelectronics	7 - Segment Displays	7 段共阳极数码管
CRYSTAL	Miscellaneous	—	晶体
POT - HG	Resistors	Variable	滑动变阻器



### 3. 实例的应用代码

实例的应用代码如例 9.2 所示。

代码在对 ATmega16 进行初始化之后即进入循环，等待比较器中断事件，在比较器中断服务程序中对比较器模块的寄存器位进行判断，根据判断的结果控制 PD 端口输出相应的字形编码。

#### 【例 9.2】 双通道模拟电压比较

```
#include <iom16v.h>
#include <macros.h>
//端口初始化
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0xFF;           //端口 D 为输出
}
//比较器初始化
void comparator_init(void)
{
    ACSR = ACSR & 0xF7;
    ACSR = 0x08;
}
//比较器中断服务子程序
#pragma interrupt_handler ana_comp_isr:iv_ANA_COMP
void ana_comp_isr(void)
{
    unsigned char temp;
    temp = ACSR;           //读寄存器状态
    if( ( temp & 0x20 ) == 0x20) //判断 ACO 位的状态, 此时 AIN0 > AIN1
    {
        PORTD = 0xC0;       //A 通道
    }
    else
    {
        PORTD = 0xF9;       //B 通道
    }
}
//ATmega16 初始化代码
```



```

void init_devices(void)
{
    CLI();
    port_init();
    comparator_init();
    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主代码
void main(void)
{
    init_devices();
    while(1)
    {
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，移动滑动变阻器，可以看到对应的比较输出，如图 9.3 所示是 AIN1 引脚上的电压高于 AIN0 引脚电压时的状态。

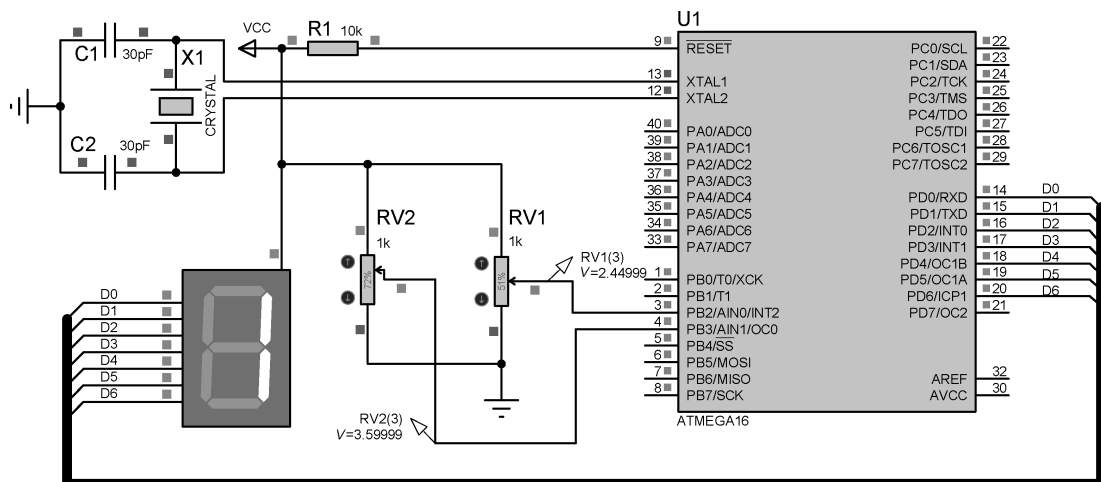


图 9.3 实例的仿真运行结果

#### 总结

实例的应用电路在 AIN0 和 AIN1 引脚上放置了两个电压探针，用于观测当前的电压值。





### 9.3.2 多通道模拟信号比较应用实例

本应用是将一个模拟信号的电压和多个模拟信号的电压进行比较，然后将比较结果通过数码管显示的实例。

#### 1. 实例的设计思路

实例的设计思路可以参考 9.3.1 节，不同的是这里使用了 ADC5（PF5）～ADC7（PF7）作为带比较信号的输入通道，该通道的选择可以参考 9.1.3 节。

#### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 9.4 所示，作为待比较电压输入的滑动变阻器 RV1 的中央抽头输出连接到 ATmega16 的 AIN0（PB2）引脚上，RV2～RV4 输出的其他待比较电压分别连接到 ADC5（PA5）～ADC7（PA7）引脚上；使用 PORTD 端口驱动了一个 7 段数码管作为显示输出，表 9.17 为电路使用的 Proteus 器件列表。

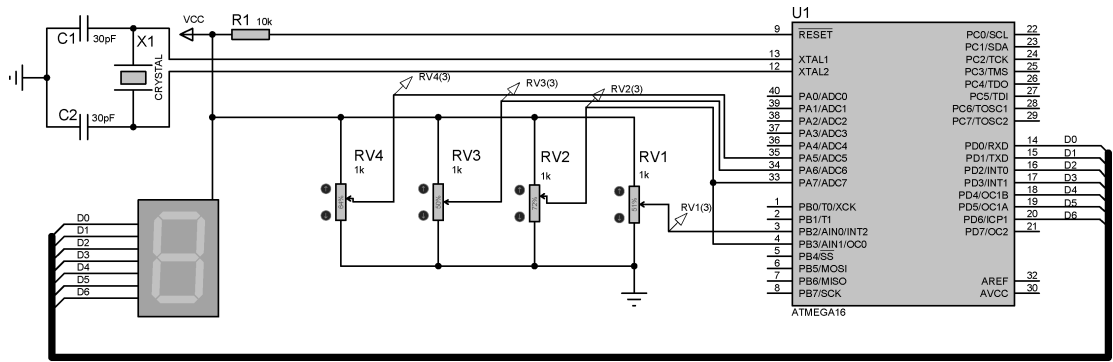


图 9.4 实例的 Proteus 电路

表 9.17 应用实例器件列表

器件名称	大 类 库	子 类 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
7SEG - COM - ANODE	Optoelectronics	7 - Segment Displays	7 段共阳极数码管
CRYSTAL	Miscellaneous	—	晶体
POT - HG	Resistors	Variable	滑动变阻器

#### 3. 实例的应用代码

实例的应用代码如例 9.3 所示，代码先对 ATmega16 的比较器模块进行初始化，然后比较 AIN0 和 AIN1 引脚上的电压值，再置位 SFIOR 寄存器的对应位，然后控制 ADMUX 切换通道对 ADC5～ADC7 引脚上的电压和 AIN0 上的电压进行比较。

#### 【例 9.3】多通道模拟电压比较

```

#include <iom16v.h>
#include <macros.h>
unsigned char const
SEGTable[ ] = { 0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x90, 0x88, 0x83, 0xc6, 0xa1,
0x86, 0x8e };
void delay( void)                //延时函数
{
    unsigned int i;
    for( i = 1; i < 100; i ++ );
}
void delay_1ms( void)            //1ms 延时函数
{
    unsigned int i;
    for( i = 1; i < ( unsigned int )( 8 * 143 - 2 ); i ++ );
}
void DelayMs( unsigned int time) //ms 延时函数
{
    unsigned int i = 0;
    while( i < time )
    {
        delay_1ms( );
        i ++ ;
    }
}
void port_init( void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0xFF;
}
//比较器初始化
void comparator_init( void)
{
    ACSR = ACSR & 0xF7;           //在初始化比较器之前必须关闭中断
    ACSR = 0x00;
}
//ATmega16 初始化

```



```

void init_devices( void)
{
    CLI();
    port_init();
    comparator_init();
    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}
//主函数
void main( void)
{
    unsigned char temp;
    init_devices();
    while(1)
    {
        DelayMs(1000);
        SFIOR = SFIOR & 0xF7;           //选择 AIN1
        temp = ACSR;                     //获得 ACSR 寄存器状态
        if( (temp & 0x20) == 0x20)      //判断状态 AIN0 > ADC5
        {
            PORTD = SEGtable[0];       //0
        }
        else
        {
            PORTD = SEGtable[1];       //1
        }
        SFIOR = SFIOR | 0x08;           //选择多工通道
        ADCSRA = 0x00;                  //关闭 ADC
        DelayMs(1000);
        ADMUX = 0x05;                   //选择 ADC5 通道
        temp = ACSR;                     //获得 ACSR 寄存器状态
        if( (temp & 0x20) == 0x20)      //判断状态 AIN0 > ADC5
        {
            PORTD = SEGtable[0];       //0
        }
        else
        {
            PORTD = SEGtable[5];       //5
        }
        DelayMs(1000);
    }
}

```

```

ADMUX = 0x06;           //选择 ADC6 通道
temp = ACSR;             //获得 ACSR 寄存器状态
if( ( temp & 0x20 ) == 0x20) //判断状态 AIN0 > ADC5
{
    PORTD = SEGtable[ 0 ]; //0
}
else
{
    PORTD = SEGtable[ 6 ]; //6
}

DelayMs( 1000 );
ADMUX = 0x07;           //选择 ADC7 通道
temp = ACSR;             //获得 ACSR 寄存器状态
if( ( temp & 0x20 ) == 0x20) //判断状态 AIN0 > ADC5
{
    PORTD = SEGtable[ 0 ]; //0
}
else
{
    PORTD = SEGtable[ 7 ]; //7
}
}

```

#### 4. 实例的仿真结果和说明

点击运行，移动滑动变阻器，可以看到对应的比较输出，如图 9.5 所示。

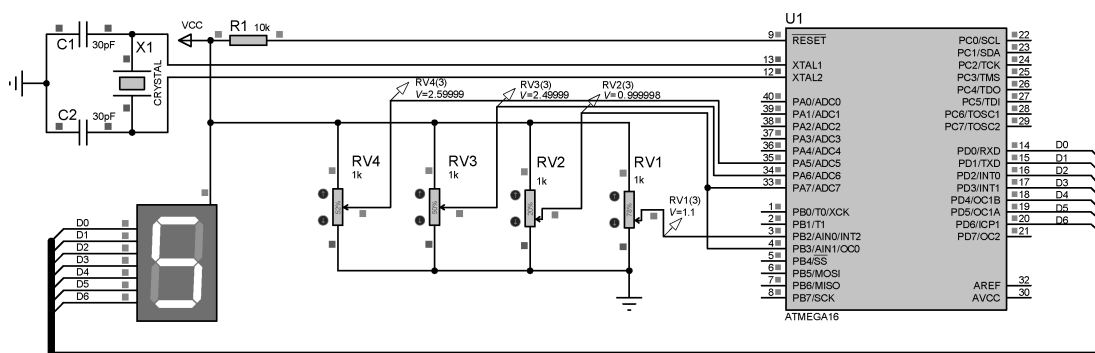


图 9.5 实例的仿真运行结果

#### 总结

在使用 ATmega16 的 ADC 输入引脚作为模拟比较器的输入引脚时，必须确保 ADC 模块已经关闭。



## 9.4 ATmega16 ADC 模块的应用实例

### 9.4.1 单通道模拟信号采集实例

本应用是使用 ATmega16 的 ADC 模块进行单通道模拟信号数据转换的实例，转换之后的结果从串行口送出。

#### 1. 实例的设计思路

实例可以通过对 ADCSRA 寄存器的状态进行查询，然后判断 ADC 转换是否完成，如果完成，则可以将结果送出。

#### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 9.6 所示，通过滑动变阻器 RV1 分压的电压连接到 ATmega16 的 ADC0 (PA0) 引脚上作为带采集模拟信号输入。

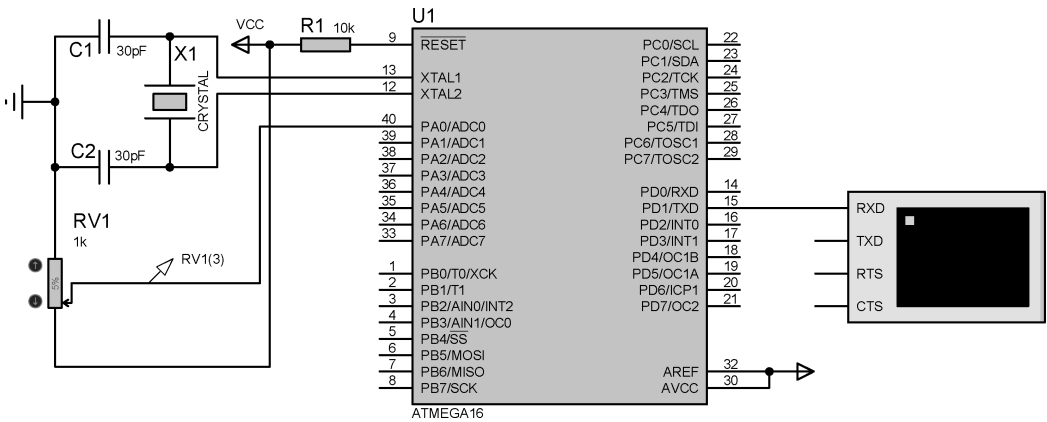


图 9.6 实例的 Proteus 电路

实例涉及的 Proteus 器件如表 9.18 所示。

表 9.18 应用实例器件列表

器件名称	大类库	子类库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
POT - HG	Resistors	Variable	滑动变阻器

#### 3. 实例的应用代码

实例的应用代码如例 9.4 所示，代码先对 ADC 模块进行初始化，然后在循环中检查 ADCSRA 寄存器中的对应位状态，如果转换完成，则将对应的采样值存放在两个 unsigned



char 类型变量 ADH、ADL 中，然后调用 putchar 函数送出。

#### 【例 9.4】 单通道模拟信号采集

```
#include <iom16v.h>
#include <macros.h>
void delay(void)                //延时函数
{
    unsigned int i;
    for(i = 1; i < 100; i++);
}
void delay_1ms(void)            //1ms 延时函数
{
    unsigned int i;
    for(i = 1; i < (unsigned int)(8 * 143 - 2); i++);
}
void delay_ms(unsigned int time) //ms 延时函数
{
    unsigned int i = 0;
    while(i < time)
    {
        delay_1ms();
        i++;
    }
}
//putchar 串口发送函数
extern void putchar(char inputdata)
{
    while(!(UCSRA & (1 << UDRE)));
    UDR = inputdata;
}
//端口初始化函数
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
}
```



```
//UART 初始化函数,9600 波特率
void uart0_init( void)
{
    UCSRB = 0x00;
    UCSRA = 0x00;
    UCSRC = BIT( URSEL) | 0x06;
    UBRRL = 0x33;           //设置波特率高低位
    UBRRH = 0x00;
    UCSRB = 0x18;
}
//ADC 初始化
void adc_init( void)
{
    ADCSR = 0x00;           //关闭 ADC
    ADMUX = 0x00;           //选择 ADC 输入通道
    ACSR   = 0x80;
    ADCSR = 0x87;
}
//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    uart0_init();
    adc_init();
    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}
//主函数
void main( void)
{
    unsigned char ADH,ADL,temp;
    init_devices();
    ADCSRA |= BIT(6);       //启动
    while(1)
    {
        temp = ADCSRA;
        if( ( temp & 0x10) == 0x10) //如果转换完成
        {
```

```

ADL = ADCL;
ADH = ADCH;           //读取数据
putchar(ADH);
putchar(ADL);
ADCSRA |= BIT(6);     //启动
delay_ms(500);
}
}
}

```

#### 4. 实例的仿真结果和说明

点击运行，调节滑动变阻器，可以在虚拟终端上看到相应的采集数据输出，如图 9.7 所示。



图 9.7 实例的仿真运行结果

### 9.4.2 多通道模拟信号采集实例

本应用是使用 ATmega16 的 ADC 模块轮询方式对多个模拟信号进行采集的实例。

#### 1. 实例的设计思路

在 9.4.1 节实例基础上，增加相应的通道切换，则可以实现多通道数据采集。

#### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 9.8 所示，RV1 ~ RV4 滑动变阻器分压的输出电压两两连接到一起，然后连接到 ADC0 (PA0) ~ ADC7 (PA7) 引脚上，实例涉及的器件列表和 9.4.1 节中的实例完全相同。

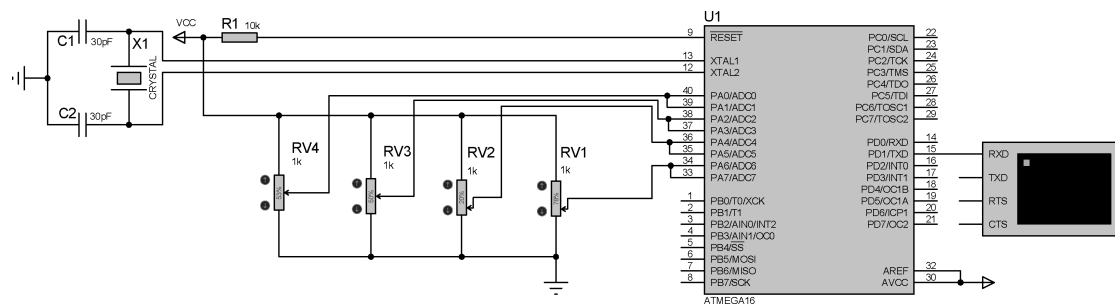


图 9.8 实例的 Proteus 电路





### 3. 实例的应用代码

实例的应用代码如例 9.5 所示, ATmega16 在初始化 ADC 模块之后进入一个 for 循环, 对 ADC0 ~ ADC7 进行轮流采样, 通过对 ADCSRA 寄存器的对应位的判断确定 ADC 转换是否完成, 和例 9.4 不同的是, 如果没有完成则等待, 然后切换到下一个通道。

#### 【例 9.5】多通道模拟信号采集

```
#include <iom16v.h>
#include <macros.h>

void delay(void)                //延时函数
{
    unsigned int i;
    for(i = 1; i < 100; i++);
}

void delay_1ms(void)            //1ms 延时函数
{
    unsigned int i;
    for(i = 1; i < (unsigned int)(8 * 143 - 2); i++);
}

void delay_ms(unsigned int time) //ms 延时函数
{
    unsigned int i = 0;
    while(i < time)
    {
        delay_1ms();
        i++;
    }
}

//putchar 串口发送函数
extern void putchar(char inputdata)
{
    while(!(UCSRA & (1 << UDRE)));
    UDR = inputdata;
}

//端口初始化函数
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
```

```

    DDRC  = 0x00;
    PORTD = 0x00;
    DDRD  = 0x00;
}
//UART 初始化函数,9600 波特率
void uart0_init( void)
{
    UCSRB = 0x00;
    UCSRA = 0x00;
    UCSRC = BIT( URSEL) | 0x06;
    UBRRL = 0x33;           //设置波特率高低位
    UBRRH = 0x00;
    UCSRB = 0x18;
}
//ADC 初始化
void adc_init( void)
{
    ADCSR = 0x00;           //关闭 ADC
    ADMUX = 0x00;           //选择 ADC 输入通道
    ACSR  = 0x80;
    ADCSR = 0x87;
}
//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    uart0_init();
    adc_init();
    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}
//主函数
void main( void)
{
    unsigned char ADH[8],ADL[8];
    unsigned char temp,i;
    init_devices();
    ADCSRA |= BIT(6);       //启动

```

```

while(1)
{
    for(i=0;i<8;i++)
    {
        while((temp & 0x10) != 0x10)           //如果转换没有完成
        {
            temp = 0x00;                       //清除
            temp = ADCSRA;
        }

        ADL[i] = ADCL;
        ADH[i] = ADCH;                         //读取数据
        putchar(ADH[i])
        putchar(ADL[i])
        ADMUX = i;
        ADCSRA |= BIT(6);                     //启动
        delay_ms(100);
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，调节滑动变阻器，可以在虚拟终端上看到相应的采集数据输出，如图 9.9 所示。

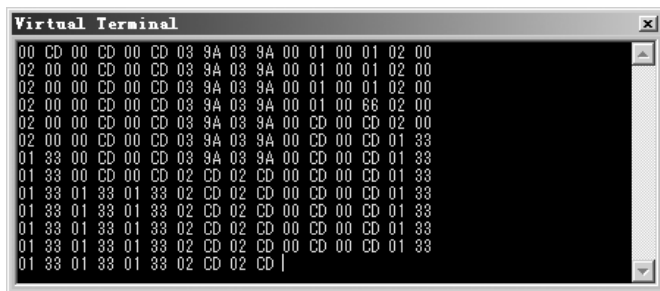


图 9.9 实例的仿真运行结果



#### 总结

多通道 ADC 采集的实质是内部一个多路开关的切换过程，所以其采集的过程是有延时的，也就是说，通道 0 和通道 7 并不是真正意义上的同时采集，如果模拟电压变化很快，则需要注意这个同步问题。

#### 9.4.3 增益放大模拟信号采集实例

在实际应用中，输入的模式信号电压值可能会比较小，此时为了不浪费采集精度，可以



将该模拟信号通过一个放大器之后再连接到 ADC 转换通道；ATmega16 的 ADC 转换模块自带一个增益放大器，本实例是使用其对一个小信号进行采样的实例。

### 1. 实例的设计思路

通过对寄存器 ADMUX 的设置，选择 ATmega16 的增益通道，然后进行采样即可。

### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 9.10 所示，通过 R2 的分压，在 RV1 的中央抽头上产生一个很小的电压，然后送到 ATmega16 的 ADC1（PA1）通道经过 10 倍放大之后进行采样，其中涉及的 Proteus 器件和 9.4.1 节中的实例完全相同。

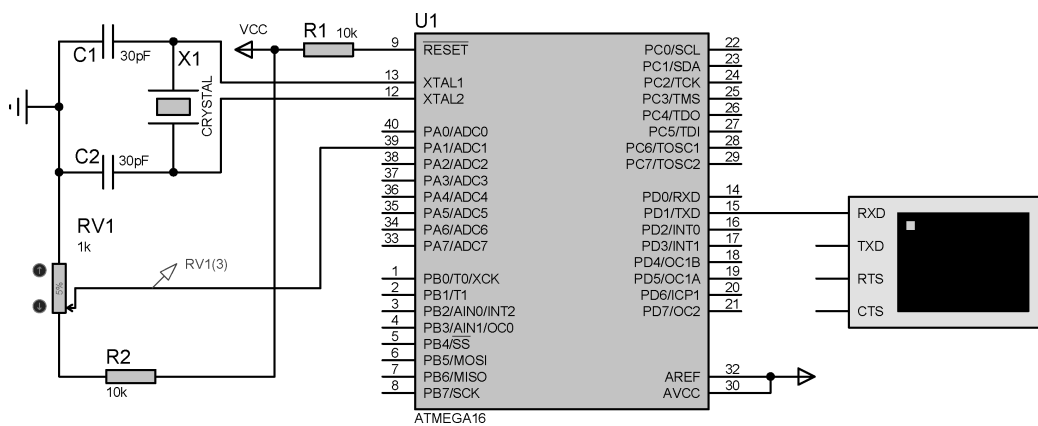


图 9.10 实例的 Proteus 电路

### 3. 实例的应用代码

实例的应用代码如例 9.6 所示，ATmega16 的 ADC 模块使用中断工作方式，当 ADC 转换完成之后进入中断服务子程序，在中断服务程序中获取 ADC 的采样值，并且将对应采集完成标志位置位，在主函数中检测这个标志位并且发送采样数据。

#### 【例 9.6】增益放大模拟电压采集

```
#include <iom16v.h>
#include <macros.h>
unsigned char ADH,ADL;           //AD 转换值
unsigned char flg=0x00;          //AD 转换标志位
void delay(void)                 //延时函数
{
    unsigned int i;
    for(i=1;i<100;i++);
}
void delay_1ms(void)             //1ms 延时函数
{
    unsigned int i;
```



```

for(i = 1; i < (unsigned int)(8 * 143 - 2); i++) ;
}
void delay_ms(unsigned int time)      //ms 延时函数
{
    unsigned int i = 0;
    while( i < time)
    {
        delay_1ms();
        i++;
    }
}
//putchar 串口发送函数
extern void putchar( char inputdata)
{
    while( ! (UCSRA & (1 << UDRE))) ;
    UDR = inputdata;
}
//端口初始化函数
void port_init(void)
{
    PORTA = 0x00;
    DDRA = 0x00;
    PORTB = 0x00;
    DDRB = 0x00;
    PORTC = 0x00;
    DDRC = 0x00;
    PORTD = 0x00;
    DDRD = 0x00;
}
//UART 初始化函数, 9600 波特率
void uart0_init(void)
{
    UCSRB = 0x00;
    UCSRA = 0x00;
    UCSRC = BIT(URSEL) | 0x06;
    UBRR1L = 0x33;                //设置波特率高低位
    UBRR1H = 0x00;
    UCSRB = 0x18;
}

//ADC 初始化函数
void adc_init(void)

```

```

{
    ADCSRA = 0x00;           //禁用 ADC
    ADMUX = 0x09;            //ADC0 输入, 10 × 放大
    ACSR = 0x80;
    ADCSRA = 0xEE;           //自动启动
}

//ADC 中断服务子函数
#pragma interrupt_handler adc_isr:iv_ADC
void adc_isr( void)
{
    ADL = ADCL;               //读出 AD 采样的值
    ADH = ADCH;
    flg = 0xff;               //置位标志位
}

//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    uart0_init();
    adc_init();
    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主函数
void main( void)
{
    init_devices();
    while( 1)
    {
        delay_ms( 500);
        if( flg == 0xff)       //如果 ADC 采样完成标志位被置位
        {
            flg = 0x00;        //清除标志位
            putchar( ADH);      //送出采样数据
            putchar( ADL);
        }
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，调节滑动变阻器，可以在虚拟终端上看到相应的采集数据输出，如图 9.11 所示。

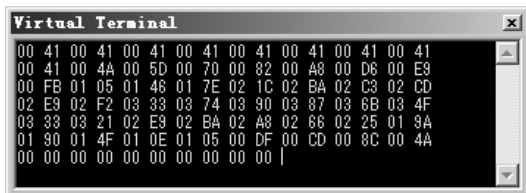


图 9.11 实例的仿真运行结果



#### 总结

在进行增益放大时，需要特别的注意寄存器 ADMUX 的设置，其余操作方法和普通采样完全相同。

#### 9.4.4 差分模拟信号比较采集实例

除了对小信号进行采样之外，在某些应用中需要采集两个模拟信号的电压差值，此时可以应用 ATmega16 模块自带的差分信号比较功能，本应用是对两个模拟信号差值采样的实例。

##### 1. 实例的设计思路

和增益放大模拟信号采样实例类似，通过对寄存器 ADMUX 的设置，选择 ATmega16 的差分通道，然后进行采样即可。

##### 2. 实例的 Proteus 电路图

实例的 Proteus 电路如图 9.12 所示，RV1 和 RV2 分压之后的两个模拟信号分别连接到 ATmega16 的 ADC0 (PA0) 和 ADC1 (PA1) 引脚，其中涉及的 Proteus 器件和 9.4.1 小节中的实例完全相同。

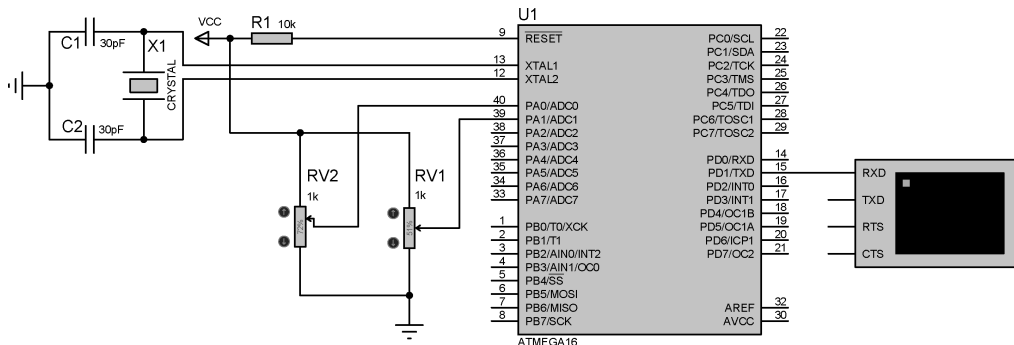


图 9.12 实例的 Proteus 电路



### 3. 实例的应用代码

实例的应用代码如例 9.7 所示, 其代码的使用和例 9.6 类似, 主要差别是 ADMUX 寄存器的选择。

#### 【例 9.7】 差分放大模拟电压采集

```
#include <iom16v.h>
#include <macros.h>
void delay(void)           //延时函数
{
    unsigned int i;
    for(i = 1; i < 100; i++);
}
void delay_1ms(void)       //1ms 延时函数
{
    unsigned int i;
    for(i = 1; i < (unsigned int)(8 * 143 - 2); i++);
}
void delay_ms(unsigned int time) //ms 延时函数
{
    unsigned int i = 0;
    while(i < time)
    {
        delay_1ms();
        i++;
    }
}
//putchar 串口发送函数
extern void putchar(char inputdata)
{
    while(!(UCSRA & (1 << UDRE)));
    UDR = inputdata;
}
//端口初始化函数
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x00;
```





```

PORTD = 0x00;
DDRD  = 0x00;
}
//UART 初始化函数, 9600 波特率
void uart0_init( void)
{
    UCSRB = 0x00;
    UCSRA = 0x00;
    UCSRC = BIT( URSEL) | 0x06;
    UBRRL = 0x33;           //设置波特率高低位
    UBRRH = 0x00;
    UCSRB = 0x18;
}

//ADC 模块初始化
void adc_init( void)
{
    ADCSRA = 0x00;           //首先关闭 ADC
    ADMUX = 0x10;           //差分输入, ADC0 为正, ADC1 为负
    ACSR  = 0x80;           //关闭模拟比较器
    ADCSRA = 0x87;           //设置 ADC 寄存器
}

//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    uart0_init();
    adc_init();
    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主程序
void main( void)
{
    unsigned char ADH, ADL, temp;
    init_devices();
    ADCSRA |= BIT(6);        //启动

```

```

while(1)
{
    delay_ms(200);
    temp = ADCSRA;
    if((temp & 0x10) == 0x10) //如果转换完成
    {
        ADL = ADCL;
        ADH = ADCH;           //读取数据
        putchar(ADH);
        putchar(ADL);
        ADCSRA |= BIT(6);     //启动
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，分别调节 RV1 和 RV2 两个滑动变阻器，可以在虚拟终端上看到相应的采集数据输出，如图 9.13 所示。

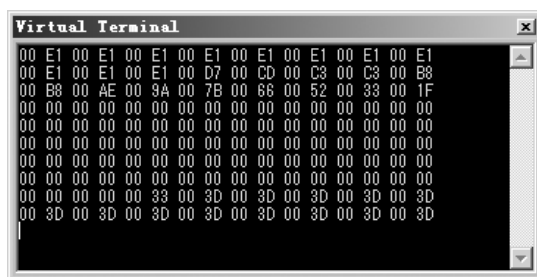


图 9.13 实例的仿真运行结果



#### 总结

在使用差分信号采集时，一定要注意“正”、“负”端的选择，也就是两个输入引脚上哪一个模拟信号电压应该更高，否则会出现采集数据全为 0 的情况。

## 第10章 ATmega16 的其他内部资源

ATmega16 还提供了看门狗和内部的 E<sup>2</sup>PROM 以供用户使用，前者用于监视系统是否正常运行，后者可以用于存放需要在掉电后保存的数据。

### 10.1 看门狗 (WDT)

ATmega16 内部集成了一个看门狗 (Watch Dog Timer) 定时器，可以用于检测上电，并且防止用户程序的“跑飞”进入死循环。

#### 10.1.1 看门狗基础

ATmega16 的看门狗定时器由独立的 1MHz 片内振荡器驱动，这是当 ATmega16 的电源为 5V 时的典型值，通过设置看门狗定时器的预分频器可以调节看门狗复位的时间间隔。

看门狗复位指令 WDR 用来复位看门狗定时器。此外，当禁止看门狗定时器或 ATmega16 发生复位时，定时器也被复位。复位时间有 8 个选项，如果没有及时复位定时器，一旦时间超过复位周期，ATmega16 将被强行复位，并执行复位向量指向的程序，看门狗具体的复位时序如图 10.1 所示。

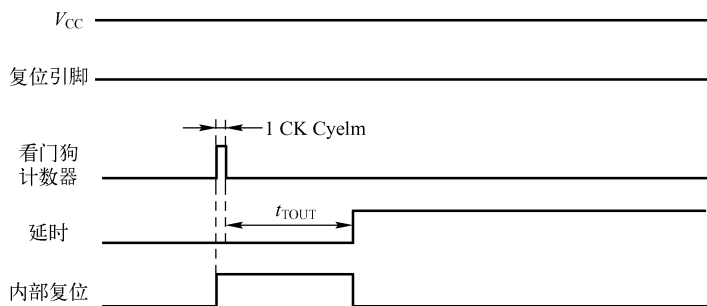


图 10.1 看门狗复位时序

#### 10.1.2 看门狗的寄存器

对 ATmega16 的看门狗的控制也是通过对寄存器的操作来完成的，WDTCR 寄存器的内部结构如表 10.1 所示。



表 10.1 看门狗定时器控制寄存器 WDTCSR

BITS	—	—	—	WDTOE	WDE	WDP2	WDP1	WDPO
读/写	R	R	R	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- **WDTOE**：关闭看门狗使能位。在清零 WDE 位的同时必须置位 WDTOE，否则不能禁止看门狗，一旦 WDTOE 被置位，ATmega16 将在紧接的 4 个时钟周期之后将 WDE 位清零。
- **WDE**：看门狗使能位。当 WDE 位被置“1”时，看门狗被使能，否则看门狗将被禁止。只有在 WDTOE 位置被置“1”时 WDE 位才能清零。
- **WDP2 ~ WDPO**：看门狗定时器预分频器位，用于决定看门狗定时器的预分频器，其预分频值及相应的溢出周期如表 10.2 所示。

表 10.2 看门狗定时器预分频

WDP2	WDP1	WDPO	WDT 振荡器周期	$V_{CC} = 3.0V$ 的溢出周期	$V_{CC} = 5.0V$ 的溢出周期
0	0	0	16K (16.384)	17.1ms	16.3ms
0	0	1	32K (32.768)	34.3ms	32.5ms
0	1	0	64K (65.536)	68.5ms	65ms
0	1	1	128K (131.072)	0.14s	0.13s
1	0	0	256K (262.144)	0.27s	0.28s
1	0	1	512K (524.288)	0.55s	0.52s
1	1	0	1.024K (1.048.576)	1.1s	1.0s
1	1	1	2.048K (2.0107.152)	2.2s	2.1s

### 10.1.3 看门狗的启动和关闭

当设置 WDE 位为“1”时，可以启动看门狗，如果要关闭看门狗，则需要按照如下步骤进行操作。

- (1) 在同一个指令内对 WDTOE 和 WDE 写“1”，即使 WDE 位已经为“1”；
- (2) 在紧接的 4 个时钟周期之内对 WDE 写“0”。

## 10.2 内部 E<sup>2</sup>PROM

ATmega16 内部包含 512B 的 E<sup>2</sup>PROM 数据存储器，作为一个独立的数据空间而存在，可以按字节寻址来读写，该 E<sup>2</sup>PROM 的寿命至少为 100 000 次擦除周期，其访问由地址寄存器、数据寄存器和控制寄存器决定。E<sup>2</sup>PROM 常常用于在 ATmega16 掉电的情况下进行数据保存，以供重新上电时调用。



### 10.2.1 E<sup>2</sup>PROM 的操作

#### 1. E<sup>2</sup>PROM 的操作时间

E<sup>2</sup>PROM 的操作时间如表 10.3 所示，用户可以使用其自定时功能来实现对它的读写操作，在操作中需要注意如下问题。

- 在电源滤波时间常数比较大的电路中，上电/掉电时  $V_{CC}$  的上升/下降速度会比较慢，此时 ATmega16 可能工作在低于晶振所要求的电源电压，需要避免 E<sup>2</sup>PROM 数据丢失。
- 为了防止无意识的 E<sup>2</sup>PROM 写操作，需要执行一个特定的写时序。
- 当执行 E<sup>2</sup>PROM 读操作时，ATmega16 会停止工作 4 个周期，然后再执行后续指令。
- 执行 E<sup>2</sup>PROM 写操作时，ATmega16 会停止工作 2 个周期，然后再执行后续指令。

表 10.3 E<sup>2</sup>PROM 操作时间

符 号	校准 RC 周期数	典型的编程时间
E <sup>2</sup> PROM 写操作	8448	8.5ms



#### 说明

时钟频率为 1MHz，不依赖于 CKSEL 的设置。

#### 2. E<sup>2</sup>PROM 的写操作

ATmega16 对 E<sup>2</sup>PROM 的写时序如下，第（3）步和第（4）步的次序可以互换。

- （1）等待 EEW<sub>E</sub> 位变为零。
- （2）等待 SPMCSR 中的 SPMEN 位变为零。
- （3）将新的 E<sup>2</sup>PROM 地址写入 EEAR 寄存器（可选）。
- （4）将新的 E<sup>2</sup>PROM 数据写入 EEDR 寄存器（可选）。
- （5）向 EECR 寄存器的 EEMWE 位写入“1”，同时清零 EEW<sub>E</sub> 位。
- （6）在 EEMWE 置位的 4 个周期内，EEW<sub>E</sub> 置位。

如果在步骤（5）和步骤（6）之间发生了中断，写操作将失败。因为，此时 E<sup>2</sup>PROM 写使能操作将超时。如果一个操作 E<sup>2</sup>PROM 的中断打断了另一个 E<sup>2</sup>PROM 操作，EEAR 或 EEDR 寄存器可能被修改，引起 E<sup>2</sup>PROM 操作失败。建议此时关闭全局中断标志 I。经过写访问时间之后，EEW<sub>E</sub> 硬件会清零，用户可以凭借这一位判断写时序是否已经完成。当 EEW<sub>E</sub> 置位后，ATmega16 要停止两个时钟周期才会运行下一条指令。

在 ATmega16 进行写 FLASH 存储器操作时，不能对 E<sup>2</sup>PROM 进行操作，所以在启动 E<sup>2</sup>PROM 写操作之前，用户软件必须检查 FLASH 的写操作是否已经完成，但是仅在软件包含引导程序并允许 ATmega16 对 FLASH 进行操作时才有用，如果 ATmega16 永远都不会写 FLASH 存储器，这一步可以省略。

#### 3. E<sup>2</sup>PROM 的掉电操作

若 ATmega16 的程序执行掉电指令时 E<sup>2</sup>PROM 的写操作正在进行，写操作将继续并在指



定的写访问时间之前完成，但是当写操作结束后，振荡器还将继续运行，芯片并非处于完全的掉电模式，因此在执行掉电指令之前应结束 E<sup>2</sup>PROM 的写操作。

若 ATmega16 的电源电压过低，E<sup>2</sup>PROM 有可能工作不正常，造成 E<sup>2</sup>PROM 数据的丢失，这种情况在使用独立的器件时也会遇到，因此需要使用相同的保护方案。由于电压过低造成 E<sup>2</sup>PROM 数据损坏有两种情况。

(1) 电压低于 E<sup>2</sup>PROM 写操作所需要的最低电压。

(2) ATmega16 本身已经无法正常工作。

E<sup>2</sup>PROM 数据丢失的问题通过当电压过低时保持 ATmega16 复位信号为低来解决，这可以通过使能芯片的掉电检测电路 BOD 来实现。如果 BOD 电平无法满足要求，则可以使用外部复位电路，在使用了这样的电路之后若写操作过程中发生了复位，只要电压足够高，写操作仍将正常结束。

## 10.2.2 E<sup>2</sup>PROM 的寄存器

ATmega16 通过对相关寄存器的控制来完成对 E<sup>2</sup>PROM 的相关操作，这些寄存器包括地址寄存器（EEARH 和 EEARL）、数据寄存器（EEDR）和控制寄存器（EECR）。

### 1. 地址寄存器 EEARH 和 EEARL

ATmega16 的 E<sup>2</sup>PROM 地址是线性增长的，其地址寄存器 EEARH 和 EEARL 用于存放需要寻址的共 512B 的地址空间，地址编码为 0 ~ 511，其内部结构如表 10.4 所示。需要注意的是，地址寄存器没有初始值，需要在访问 E<sup>2</sup>PROM 之前设置。

表 10.4 地址寄存器 EEARH 和 EEARL

EEARH	—	—	—	—	—	—	EEAR9	EEAR8
EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
读/写	R	R	R	R	R	R	R/W	R/W
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	X	X
	X	X	X	X	X	X	X	X

### 2. 数据寄存器 EEDR

对于 E<sup>2</sup>PROM 写操作，EEDR 寄存器里是需要写到 EEAR 单元的数据，对于读操作，EEDR 寄存器是从地址 EEAR 读取的数据，其内部结构如表 10.5 所示。

表 10.5 数据寄存器 EEDR

EEDR	MSB	—	—	—	—	—	—	LSB
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

### 3. 控制寄存器 EECR

控制寄存器 EECR 用于对 E<sup>2</sup>PROM 的操作，其内部结构如表 10.6 所示。



表 10.6 数据寄存器 EEDR

EEDR	—	—	—	—	EERIE	EEMWE	EEWE	EERE
读/写	R	R	R	R	R/W	R/W	R/W	R/W
初始值	0	0	0	0	0	0	0	0

- **EERIE**: E<sup>2</sup>PROM 中断使能位。若 SREG 寄存器的 I 标志被置“1”，置位 EERIE 标志位将使能 E<sup>2</sup>PROM 中断，清零 EERIE 位则禁止此中断。
- **EEMWE**: E<sup>2</sup>PROM 主机写使能位。该位用于控制 EEWE 被置“1”时是否可以启动 E<sup>2</sup>PROM 的写操作。当 EEMWE 被置“1”时，在 4 个时钟周期内 EEWE 置位将把数据写入指定地址；若 EEMWE 为“0”，则操作 EEWE 位将不起作用，在 EEMWE 被置位后 4 个周期，硬件对其清零。
- **EEWE**: E<sup>2</sup>PROM 写使能位。EEWE 位为 E<sup>2</sup>PROM 写操作的使能信号，当 E<sup>2</sup>PROM 数据和地址设置好之后，需置位 EEWE 以便将数据写入，此时 EEMWE 必须被先置位，否则写操作将不会发生。
- **EERE**: E<sup>2</sup>PROM 读使能位。该位为读操作的使能信号，当地址设置好之后，需置位 EERE 以便将数据读入 EEAR 寄存器，E<sup>2</sup>PROM 数据的读取只需要一条指令，且无须等待。读取 E<sup>2</sup>PROM 后 ATmega16 要停止 4 个时钟周期才可以执行下一条指令。用户在读取 E<sup>2</sup>PROM 时应该检测 EEWE，需要注意的是，此时如果一个写操作正在进行，就无法读取 E<sup>2</sup>PROM，也无法改变寄存器 EEAR。

### 10.2.3 E<sup>2</sup>PROM 的操作函数

ICC AVR 在 EEPROM.h 头文件中提供了 EEPROM\_READ 和 EEPROM\_WRITE 两个函数，用于对 ATmega16 的内部 E<sup>2</sup>PROM 进行操作，这些函数的说明和原型如下。

#### 1. E<sup>2</sup>PROM 读函数

EEPROM\_READ 函数用于从 E<sup>2</sup>PROM 中读出数据，其原型如例 10.1 所示，调用方式如下。

- **EEPROM\_READ(intadd, unsigned char data)**: 从地址 add 中读出输出 data。

#### 【例 10.1】E<sup>2</sup>PROM 读函数

```
unsigned char E2PROM_read(unsigned int uiAddress)
{
    /* 等待上一次写操作结束 */
    while(EECR & (1 << EEWE));
    /* 设置地址寄存器 */
    EEAR = uiAddress;
    /* 设置 EERE 以启动读操作 */
    EECR |= (1 << EERE);
    /* 自数据寄存器返回数据 */
}
```



```
    return EEDR;
}
```

## 2. E<sup>2</sup>PROM 写函数

EEPROM\_WRITE 用于将数据写入 E<sup>2</sup>PROM 中，其原型如例 10.2 所示，调用方式如下。

- EEPROM\_WRITE(intadd, unsigned char data)：将数据 data 写入地址 add 中。

### 【例 10.2】 E<sup>2</sup>PROM 写函数

```
void E2PROM_write(unsigned int uiAddress, unsigned char ucData)
{
    /* 等待上一次写操作结束 */
    while(EECR & (1 << EEWE));
    /* 设置地址和数据寄存器 */
    EEAR = uiAddress;
    EEDR = ucData;
    /* 置位 EEMWE */
    EECR |= (1 << EEMWE);
    /* 置位 EEWE 以启动写操作 */
    EECR |= (1 << EEWE);
}
```

## 10.3 内置看门狗和 E<sup>2</sup>PROM 应用实例

### 10.3.1 内置看门狗模块测试应用实例

本应用是一个对 ATmega16 内置看门狗模块的运行状态进行测试的实例。

#### 1. 实例的设计思路

ATmega16 在开启看门狗模块之后必须定时“喂狗”，以防止看门狗的溢出导致复位。本实例在 ATmega16 启动时点亮一个 LED D1，然后在系统正常运行时将这个 LED 熄灭，让另外一个 LED D2 周期性闪烁；如果当按键被按下，则停止“喂狗”，此时 ATmega16 将由看门狗溢出而导致复位重新初始化，D1 点亮后熄灭。

#### 2. 实例的 Proteus 电路

实例的 Proteus 电路如图 10.2 所示，两个发光二极管 D1 和 D2 使用灌电流驱动方式连接到 ATmega16 的 PC0 和 PC5 引脚上，一个按键 K1 连接到 ATmega16 的外部中断 0 引脚（PD2）上，实例涉及的 Proteus 器件如表 10.7 所示。



#### 注意

将和发光二极管连接的 R2 和 R3 的属性都设置为 DIGITAL（数字），此时电阻两端的电压值和其标示电阻值无关，仅仅和逻辑电平有关。





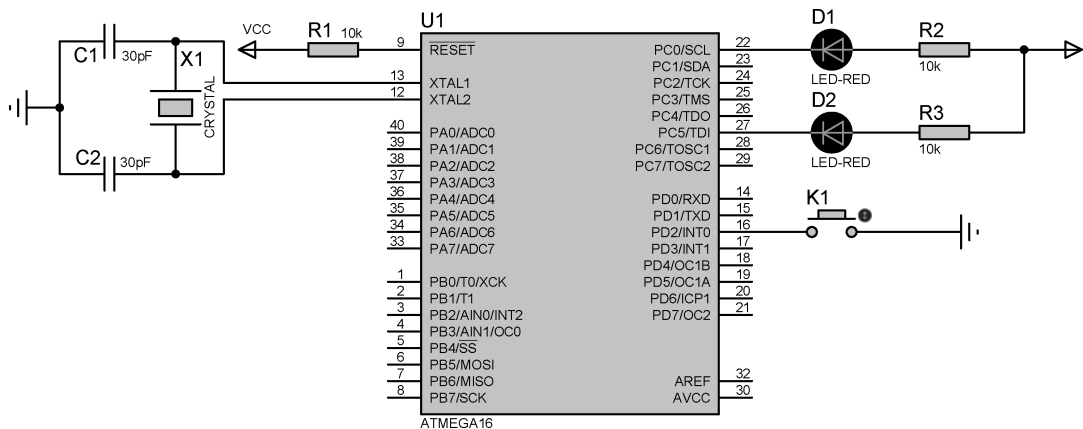


图 10.2 实例的 Proteus 电路

表 10.7 应用实例器件列表

器件名称	大 类 库	子 类 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CRYSTAL	Miscellaneous	—	晶体
BUTTON	Switches & Relays	Switches	独立按键
LED - RED	Optoelectronics	LEDs	红色发光二极管

### 3. 实例的应用代码

实例的应用代码如例 10.3 所示。

应用代码在定时器 T/C0 的中断服务子程序中进行“喂狗”操作，而在外部中断 0 的服务子程序中将定时器 T/C0 关闭，从而停止“喂狗”。

#### 【例 10.3】 内置看门狗模块测试

```
#include <iom16v.h>
#include <macros.h>

void delay(void) //延时函数
{
    unsigned int i;
    for(i = 1;i < 100;i ++ );
}

void delay_1ms(void) //1ms 延时函数
{
    ...
}
```

```

    unsigned int i;
    for(i = 1; i < (unsigned int)(8 * 143 - 2); i++) ;
}

void delay_ms(unsigned int time)                //ms 延时函数
{
    unsigned int i = 0;
    while(i < time)
    {
        delay_1ms();
        i++;
    }
}

//端口初始化
void port_init(void)
{
    PORTA = 0x00;
    DDRA  = 0x00;
    PORTB = 0x00;
    DDRB  = 0x00;
    PORTC = 0x00;
    DDRC  = 0x21;                //PC0 和 PC5 为输出
    PORTD = 0x00;
    DDRD  = 0x00;
}

//看门狗初始化, 16K 分频
void watchdog_init(void)
{
    WDTCR = 0x1F;
    WDTCR = 0x0F;                //使能看门狗
}

//关闭看门狗
void WDT_off(void)
{
    //置位 WDCE 和 WDE
    WDTCR = (1 << WDTOE) | (1 << WDE);
    //关闭 WDT
    WDTCR = 0x00;
}

//TC0 初始化函数, 10ms 溢出一次
void timer0_init(void)

```



```

{
    TCCR0 = 0x00;
    TCNT0 = 0xB2;                //设置计数器值
    OCR0  = 0x4E;
    TCCR0 = 0x05;                //启动 TC0
}

//TC0 溢出中断处理函数
#pragma interrupt_handler timer0_ovf_isr:iv_TIMO_OVF
void timer0_ovf_isr( void)
{
    TCNT0 = 0xB2;
    WDR( );                      //喂狗
}

//外部中断 0 处理子函数
#pragma interrupt_handler int0_isr:iv_INT0
void int0_isr( void)
{
    TCCR0 = 0x00;                //关闭定时器
}

//ATmega16 初始化函数
void init_devices( void)
{
    CLI( );
    port_init( );
    watchdog_init( );
    timer0_init( );

    MCUCR = 0x02;
    GICR  = 0x40;
    TIMSK = 0x01;
    SEI( );
}

//主函数
void main( void)
{
    init_devices( );
    PORTC &= ~BIT(0);
    delay_ms( 1000 );
    PORTC |= BIT(0);             //PC0 = 0, LED1 灭
    while( 1 )
    {
        delay_ms( 1000 );
    }
}

```



```
PORTC^=BIT(5);           //翻转 LED
}
}
```

#### 4. 实例的仿真结果和说明

点击运行，可以看到 D1 闪烁，如果按下按键，则 D1 会停止闪烁而 D2 闪一次。

### 10.3.2 E<sup>2</sup>PROM 读写应用实例

本应用是一个用 ATmega16 单片机的 I/O 引脚扩展一个 4×4 的行列扫描键盘，16 个按键分别对应“0”～“F”编码，使用一个数码管将被按下的按键对应编码显示出来，并且将对应的数据写入 E<sup>2</sup>PROM 中保存，当按键为“F”则一次性读出显示的实例。

#### 1. 实例的设计思路

ICC AVR 的 EEPROM.h 头文件中提供了 EEPROM\_READ 和 EEPROM\_WRITE 两个函数，用于对 ATmega16 的内部 E<sup>2</sup>PROM 进行操作。

#### 2. 实例的 Proteus 电路

实例的 Proteus 电路如图 10.3 所示，其与 5.4.5 节中的实例 5.5——计算器键盘显示是完全相同的，在按键上添加了对应的编号（0～F），其涉及的 Proteus 器件也完全相同。

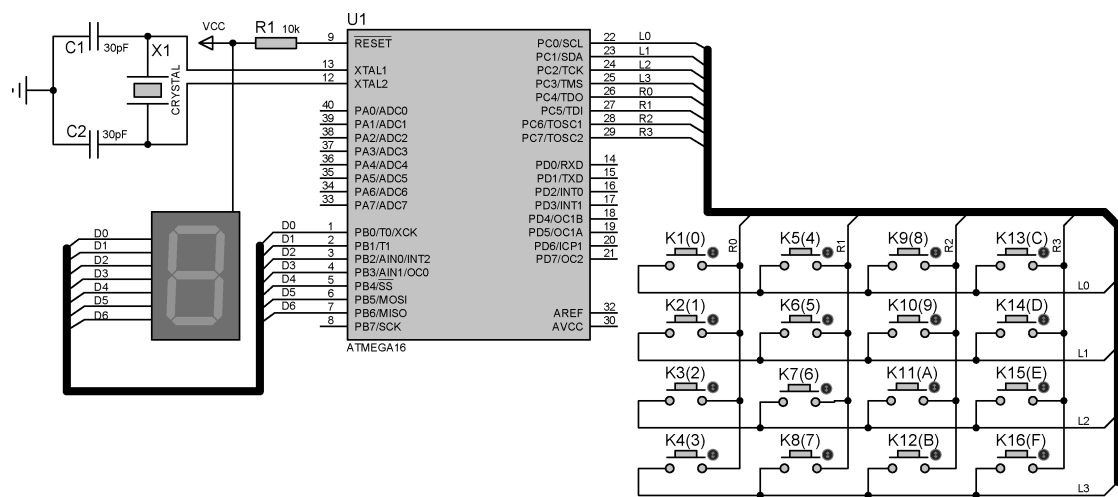


图 10.3 实例的 Proteus 电路

#### 3. 实例的应用代码

实例的代码如例 10.4 所示，其中涉及的两个用于 E<sup>2</sup>PROM 的操作的基础函数如 10.2.3 节所示。

#### 【例 10.4】 内部 E<sup>2</sup>PROM 读写测试

```
#include <iom16v.h>
#include <EEPROM.h>
```



```

unsigned int Key_num = 20;
unsigned char wData;           //待写入数据
unsigned char rData;           //读出的数据
const unsigned char SEGtable[ ] =
{
    0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,
    0x88,0x83,0xc6,0xa1,0x86,0x8e,0xbf,0xff //共阳极 0. 1. 2. 3. 4. 9. 6. 7. 8. 9. A. b. C. d. E. F. -
};
void delay( void)              //延时函数
{
    unsigned int i;
    for( i = 1;i < 100;i ++ );
}
void delay_1ms( void)          //1ms 延时函数
{
    unsigned int i;
    for( i = 1;i < ( unsigned int )( 8 * 143 - 2 );i ++ );
}
void delay_ms( unsigned int time) //ms 延时函数
{
    unsigned int i = 0;
    while( i < time)
    {
        delay_1ms( );
        i ++ ;
    }
}

void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0x00;
    PORTB = 0x00;
    DDRB = 0xFF;              //B 口全部输出驱动数码管
    PORTC = 0x00;
    DDRC = 0xf0;              //C 口低四位输入, 置高电平, 高四位输出, 置低电平
    PORTD = 0x00;
    DDRD = 0x00;
}

unsigned char Key_scan( void)   //键盘扫描函数
{
    unsigned char i,j;

```

```

DDRC = 0xf0; //设置 PD 高四位为输出口，低四位为输入口
PORTC = 0x0f; //初始运行输出全为 0x0f
if( ( PINC & 0x0f) == 0x0f) return 20; //判断有无按键动作，没有，返回 20
else
{
    delay_ms(10); //按键消抖
    if( ( PINC & 0x0f) == 0x0f) return 20; //再次判断是否有按键动作
    else
    {
        for( i = 4; i < 8; i ++ ) //逐行输出 0
        {
            PORTC = ~( 1 << i ) | 0x0f; //第 i 行输出 0
            for( j = 0; j < 4; j ++ )
            {
                if( ( PINC & ( 1 << j) ) == 0) //逐列检测
                    Key_num = ( i - 4 ) * 4 + j; //计算键值
            }
        }
        return Key_num; //返回键值
    }
}

//显示函数
void Led_display( void)
{
    PORTB = SEGtable[ Key_num ]; //查表对应的显示编码，送出
}

//初始化 ATmega16，注意不要对中断进行操作
void init_devices( void)
{
    port_init();
    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x00;
}

//主函数
void main( void)
{
    unsigned char counter = 0;
    unsigned char readdata = 0x00;
    init_devices();
    EEPROM_READ(0x00, rData); //将上一次的数据读入

```

```

PORTB = SEGtable[ rData ];           //显示读出的数据
while( 1 )
{
    Key_scan( ) ;                     //按键扫描
    Led_display( ) ;                 //显示
    wData = Key_num ;                 //将按键值送给待写入数据
    EEPROM_WRITE( ( unsigned int ) ( 0x20 + wData ) , wData ); //将数据写入 E2PROM
    if( wData == 0x0F )                //F 按键被按下
    {
        for( counter = 0 ; counter < 16 ; counter ++ ) //一次读出显示
        {
            delay_ms( 500 ) ;          //延时
            EEPROM_READ( ( unsigned int ) ( counter + 0x20 ) , readdata );
            PORTB = SEGtable[ readdata ];
        }
        wData = 0x00 ;
    }
}

```

#### 4. 实例的仿真结果和说明

点击运行，按下相应按键，然后点击暂停，打开 Debug 菜单下的 AVR EPROM Memory 窗口，可以看到对应的数据，如图 10.4 所示。

AVR EPROM Memory - U1																	X																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																	
0000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF</

图 10.4 实例的仿真运行结果



#### 总结

对 ATmega16 的 E<sup>2</sup>PROM 进行读/写操作的速度比较慢，需要保留足够长的时间。

# 第11章 ATmega16 的应用系统

本章将介绍三个 ATmega16 的应用系统，分别是“单 I/O 引脚扩展多按键”、“简易电子琴”和“商场灯光控制”。

## 11.1 单 I/O 引脚扩展多按键

在基于 ATmega16 单片机的应用系统中，由于需要扩展的外部资源比较多，常常会出现 I/O 引脚不够用的情况，此时如果需要扩展多个按键，可以使用单 I/O 引脚来扩展的设计模式，从而大大减小对单片机 I/O 引脚的需求。

### 11.1.1 应用系统背景

单 I/O 引脚扩展多按键适用于按键数目较少或者特别多，用行列扫描键盘和独立按键扩展都不合适的应用场合，能大大地减小系统对于 ATmega16 单片机的 I/O 需求。从理论上来说，在这个引脚上可以扩展 0 ~ 256 个乃至 1024 个引脚，但需要注意的是，这个“单 I/O 引脚”其实不是真正意义上的“普通” I/O 引脚，需要的是一个第二功能是 ADC 模块输入端的引脚。

### 11.1.2 设计思路

#### 1. 系统工作流程

单 I/O 引脚扩展多按键应用系统的工作流程如图 11.1 所示，系统判断被按下的按键编号，然后进行相应的操作。

#### 2. 系统的需求分析与设计

设计单 I/O 引脚扩展多按键系统，需要考虑如下几个方面的内容。

- 如何辨别连接到单引脚上的各个按键。
- 需要一个能将按键对应编码显示出来的器件。
- 需要设计合适的单片机软件。

#### 3. 系统原理

单 I/O 引脚扩展多按键使用了电阻网络的连接方式，多个按键分别被连接到一个电阻网络的两端，然后连接到 ATmega16 单片机的内置 ADC 模块输入引脚上，当有按键被按下时，输入引脚上被外加上不同的电压值，通过对这

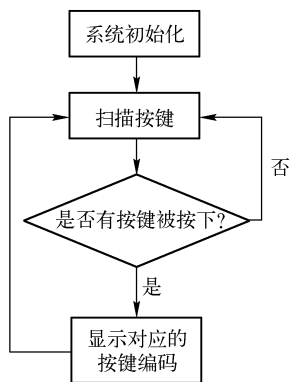


图 11.1 单 I/O 引脚扩展多按键应用系统的工作流程



个电压值的判断来确定被按下的按键编号。

如图 11.2 所示,此时可以根据 ADC 转换的判断有无按键被按下或是哪个按键被按下。根据电阻值(为方便举例,全部取值为  $1\text{ k}\Omega$ ,可以自行取其他阻值,电压平均分配就最好)可以算出:无按键被按下,电压值为  $V_{CC}$ ; S1 被按下,电压值为 0; S2 被按下,电压值为  $1/2V_{CC}$ ;同理,  $S3 = 2/3V_{CC}$ ;  $S4 = 3/4V_{CC}$ ;  $S5 = 4/5V_{CC}$ 。根据 ADC 采集值比较最接近哪个键值,就是该按键。

使用此工作原理来扩展多个按键需注意以下两点。

(1) 按键总数目不可太多,其和 ADC 模块的分辨率有关系,否则将会检测不准。以 8 位分辨率来说,共 256 点,每个按键的点最好为 25 点以上,即  $256/25 = 10$ ,最多 10 个按键,具体可以自行实验。

(2) 按键选用接触性较好的按键。按键差的使用时间一长,容易产生接触电阻,按键多容易误检测。

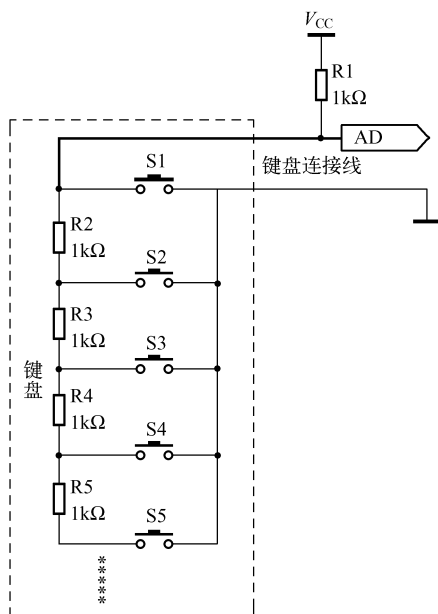


图 11.2 电阻网络连接方式

### 11.1.3 硬件系统设计

#### 1. 硬件系统模块划分

单 I/O 引脚扩展多按键应用系统的硬件模块如图 11.3 所示,由 ATmega16 单片机、按键输入网络和显示模块组成,其各部分详细说明如下。

- ATmega16 单片机:这是单 I/O 引脚扩展多按键应用系统的核心控制器。

- 按键输入网络:多按键的连接网络。
- 显示模块:显示当前按键的编码状态。

#### 2. 硬件系统电路

单 I/O 引脚扩展多按键应用系统的 Proteus 电路如图 11.4 所示,ATmega16 单片机使用 PA0 引脚(内置 ADC 模块输入通道 0)通过电阻网络扩展了 10 个独立按键作为输入,使用 PC 端口扩展了一个 7 位独立数码管作为显示模块,此外 ATmega16 的引脚 AREF 和 AVCC 均连接到  $V_{CC}$ 。

单 I/O 引脚扩展多按键应用系统硬件模块电路中涉及的典型器件如表 11.1 所示。

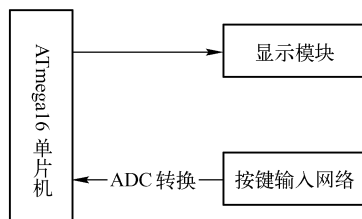


图 11.3 单 I/O 引脚扩展多按键应用系统硬件模块

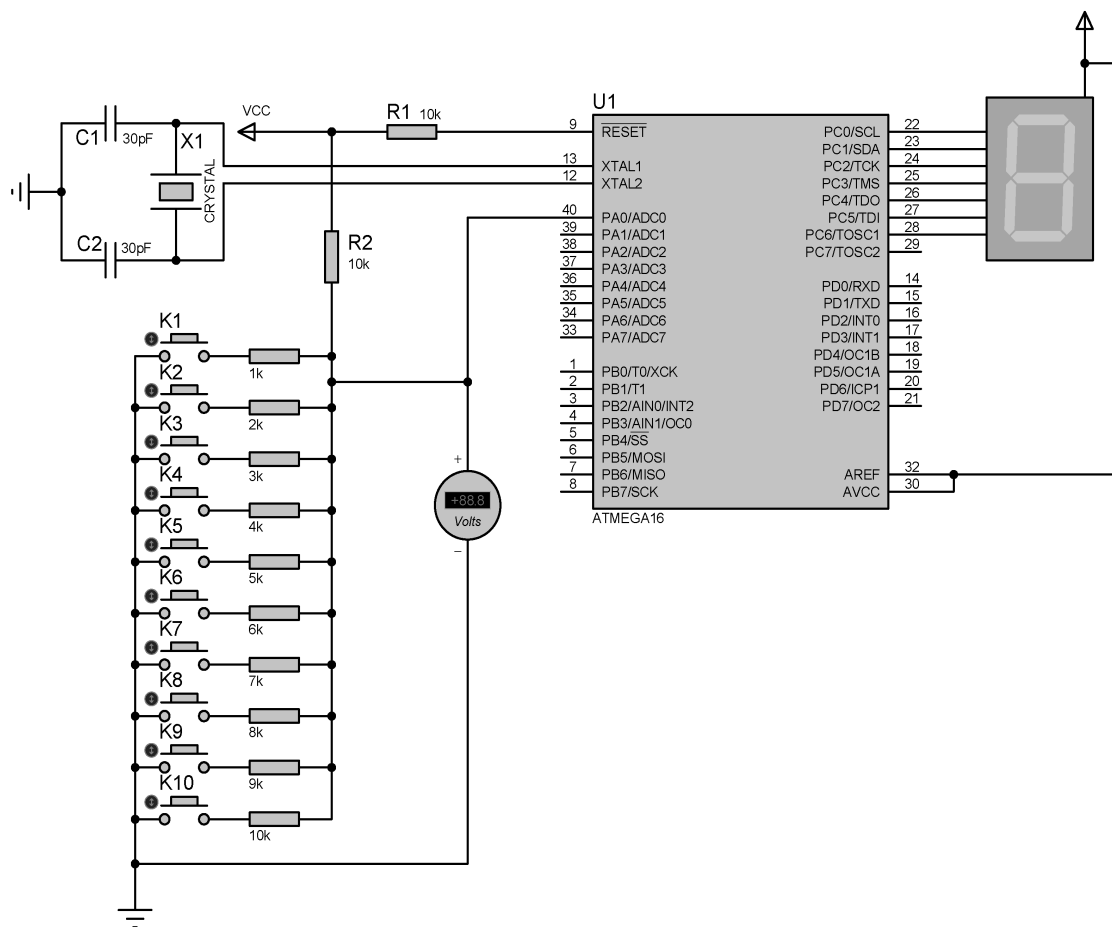


图 11.4 单 I/O 引脚扩展多按键应用系统的 Proteus 电路

表 11.1 单 I/O 引脚扩展多按键应用系统电路涉及的典型器件

器件名称	器件编号	说明
晶体	X1	ATmega16 单片机的振荡源
ATmega16 单片机	U1	ATmega16 单片机，系统的核心控制器件
电容	C1、C2、C4	滤波，储能器件
电阻	R1、R2 等	限流，上拉
单位数码管		显示器件
按键	K1 ~ K10	输入器件

### 3. 硬件系统模块介绍

应用系统主要涉及独立按键（参考 5.4.4 节）、单位数码管（参考 5.4.3 节）和 ATmega16 的 A/D 模块的使用（参考 9.2 节）。



### 11.1.4 软件系统设计

#### 1. 软件系统流程

单 I/O 引脚扩展多按键应用系统的软件工作流程如图 11.5 所示，它通过对 ADC 转换的结果来判别对应的按键，并且将其对应编号通过显示模块进行显示。

#### 2. 软件系统的应用代码

单 I/O 引脚扩展多按键应用系统的应用代码如例 11.1 所示，它使用了一个 switch 语句对当前的 ADC 采样值进行相应的判断，然后根据对应的按键值进行相应的显示。

**【例 11.1】** 单 I/O 引脚扩展多按键应用系统的应用代码

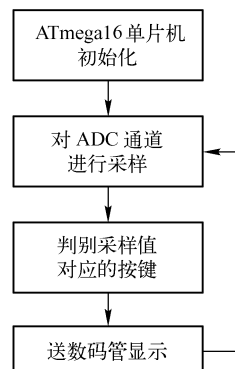


图 11.5 单 I/O 引脚扩展多按键应用系统的软件工作流程

```
#include <iom16v.h>
```

```
#include <macros.h>
```

```
unsigned char Segtable[] = {0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90};
```

```
void delay(void) //延时函数
```

```
{
    unsigned int i;
    for(i = 1;i < 100;i ++ );
}
```

```
void delay_1ms(void) //1ms 延时函数
```

```
{
    unsigned int i;
    for(i = 1;i < (unsigned int)(8 * 143 - 2);i ++ );
}
```

```
void delay_ms(unsigned int time) //ms 延时函数
```

```
{
    unsigned int i = 0;
    while(i < time)
    {
        delay_1ms();
        i ++ ;
    }
}
```

```
//端口初始化函数
```

```
void port_init(void)
```

```

{
    PORTA = 0x00;
    DDRA = 0x00;
    PORTB = 0x00;
    DDRB = 0x00;
    PORTC = 0x00;          //PC 作为输出
    DDRC = 0xFF;
    PORTD = 0x00;
    DDRD = 0x00;
}

//ADC 初始化
void adc_init( void)
{
    ADCSR = 0x00;          //关闭 ADC
    ADMUX = 0x00;          //选择 ADC 输入通道
    ACSR = 0x80;
    ADCSR = 0x87;
}

//初始化 ATmega16
void init_devices( void)
{
    CLI();
    port_init();
    adc_init();
    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x00;
    SEI();
}

//主函数
void main( void)
{
    unsigned char ADH,ADL,temp;
    unsigned int ADtemp;
    init_devices();
    ADCSRA |= BIT(6);      //启动
    while(1)
    {
        temp = ADCSRA;
        if( ( temp & 0x10) == 0x10)      //如果转换完成
    }

```



```

ADL = ADCL;
ADH = ADCH;                                //读取数据
ADtemp = ADH * 0xFF + ADL;
ADH = (ADtemp >> 2) % 0xff;                 //获得 8 位采集值
switch(ADH)                                 //根据所得值判断按键
{
    case 0x17: PORTC = Segtable[0]; break;
    case 0x2a: PORTC = Segtable[1]; break;
    case 0x3b: PORTC = Segtable[2]; break;
    case 0x48: PORTC = Segtable[3]; break;
    case 0x55: PORTC = Segtable[4]; break;
    case 0x5F: PORTC = Segtable[5]; break;
    case 0x69: PORTC = Segtable[6]; break;
    case 0x71: PORTC = Segtable[7]; break;
    case 0x79: PORTC = Segtable[8]; break;
    case 0x7F: PORTC = Segtable[9]; break;
    default: {}
}

ADCSRA |= BIT(6);                          //启动
delay_ms(50);
}
}
}

```

### 11.1.5 应用系统的仿真和总结

在 Proteus 中绘制如图 11.4 所示的电路, 其中涉及的典型器件如表 11.2 所示。

表 11.2 Proteus 电路器件列表

器件名称	库	子 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CAP - ELEC	Capacitors	Generic	极性电容
CRYSTAL	Miscellaneous	—	晶体
BUTTON	Switches & Relays	Switches	独立按键
7 - SEG - COM - ANODE	Optoelectronics	7 - Segment Displays	共阳极 7 段数码管

#### 1. Proteus 中的电压表和电流表

电压表和电流表是电路系统中最常用的测试仪器, Proteus ISIS 同样提供了对应的虚拟

工具，分别是 AC Voltmeter（交流电压表）、AC Ammeter（交流电流表）、DC Voltmeter（直流电压表）和 DC Ammeter（直流电流表），如图 11.6 所示。

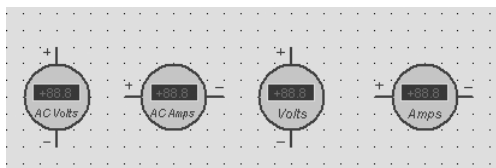


图 11.6 虚拟电压表和电流表

这 4 个电表的使用方法和实际的交流、直流电表一样，电压表并联在被测电压两端，电流表串联在电路中，要注意方向。运行仿真时，直流电表出现负值，说明电表的极性接反了，两个交流表显示的是有效值。

双击原理图中的对应电压表或者电流表，弹出如图 11.7 所示的属性设置对话框（以交流电压表为例）。

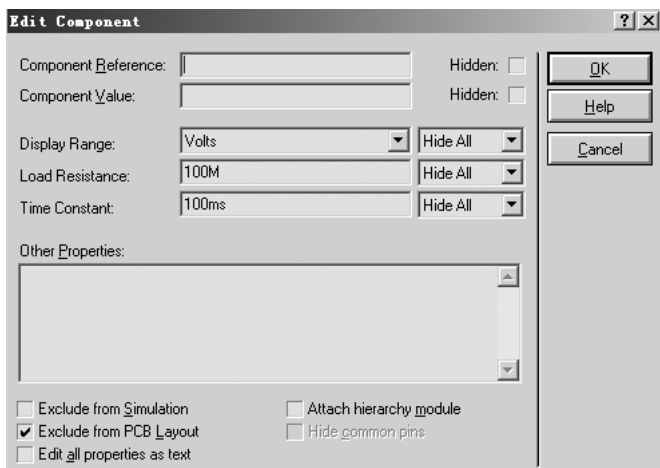


图 11.7 电压表/电流表属性设置对话框

电压表和电流表的属性说明如下。

- Display Range: 显示值，也就是对应的电压表和电流表的刻度值，电压表通常有 Volts（伏）、MilliVolts（毫伏）、MicroVolts（微伏）三个选项，类比电流表的 Amps（安培）、Milliamps（毫安）和 Microamps（微安）。
- Load Resistance: 电压表或者电流表本身的内阻大小，默认为  $100\text{M}\Omega$ ，通常来说它需要设置为一个比较大的值。
- Time Constant: 持续时间。

## 2. 单 I/O 引脚扩展多按键应用系统的仿真

点击运行，按下按键，可以看到电压表上相应的电压变化，并且看到对应的按键编码显示，如图 11.8 所示。

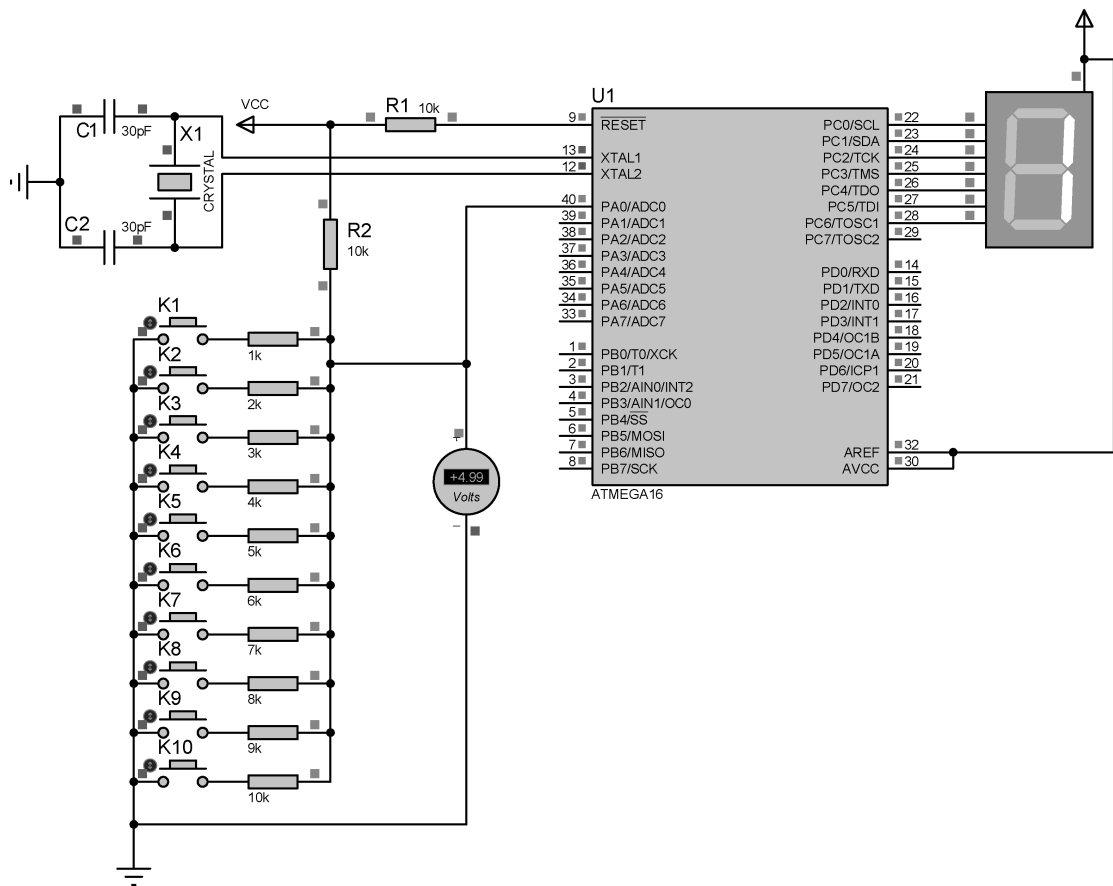


图 11.8 单 I/O 引脚扩展多按键的 Proteus 的仿真



## 总结

在实际应用中，应该合理地划分按键对应的电压区间，不能过于密集，否则会出现大量的“误码”。

## 11.2 简易电子琴

简易电子琴是一种简易的演奏乐器，它能在 ATmega16 单片机的控制下根据用户的输入发出指定的音乐效果，这种效果可以应用在各种提示音、背景音中，可以起到提示或者渲染环境气氛的作用。

### 11.2.1 应用系统背景

人类通常听到的声音可以分噪声和乐声两种，噪声是无规律的声音而乐声是有规律的声音，简易电子琴所播放的声音主要是乐声。

从人的听觉来感受，乐声有高低之分，当发声物体振动频率高时，对应的乐声就高，反之则低。简易电子琴所使用的乐声范围通常从每秒振动 16 次（最低音）到振动 4186 次

（最高音），可以划分为 97 个等级。

不同音高的乐声是用“C、D、E、F、G、A、B”这 7 个字母来表示的，它们被称为乐声的音名。在实际使用中，通常使用“do、re、mi、fa、sol、la、si”来对音名进行发声操作，其对应了简谱中的“1、2、3、4、5、6、7”。

对应的乐声持续时间称为乐声的持续时间，使用节拍数来表示。

对于一段音乐来说，它是由许多不同的音符组成的，而每个音符对应了不同的发生频率，所以简易电子琴可以使用发声系统进行不同频率的发声，并且加以和节拍数对应的延时来产生音乐。



### 注意

简谱中对应的“1、2、3、4、5、6、7”称为自然音，除了自然音之外，乐声中还存在升、降、半音等概念和分类，读者可以自行参阅相应的资料。

简易电子琴提供了一系列按键来分别对应基本的自然音，当用户按下了对应的按键时发出对应的乐声，并且提供相应的指示，此外为了演示，简易电子琴还内置了一首音乐可以完整地供用户播放试听。

## 11.2.2 设计思路

### 1. 系统工作流程

简易电子琴应用系统的工作流程如图 11.9 所示，简易电子琴应用系统在初始化完成之后等待按键被按下，当有按键被按下时首先判断按键的类型，如果是播放键，则播放预先内置的音乐，如果是演奏键，则驱动发声部件发出相应的乐声，并且给出相应的指示。

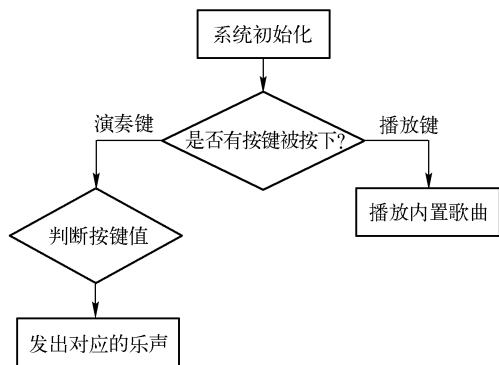


图 11.9 简易电子琴应用系统的工作流程

### 2. 系统的需求分析与设计

设计简易电子琴系统，需要考虑如下几个方面的内容。

- 要播放音乐，则需要一个能发出相应乐声的发声器件，并且使得 ATmega16 单片机能对该发声器件进行驱动。
- 能让用户进行音乐的输入，需要提供和基本音符对应的按键。
- 需要能让用户了解对应的按键已经被按下，需要有对应的指示灯。
- 要根据相应的乐声基础概念来驱动发声器件发出不同的乐声效果，需要设计合适的单片机软件。

### 3. 系统原理

由于乐声是由于不同的频率构成的，所以可以使用 ATmega16 单片机的定时器来产生不同的脉冲驱动发声器件，即可得到对应的音符。





假设 ATmega16 单片机工作时钟为 8MHz，使用定时计数器 T/C1 的工作方式 1 来进行定时操作，其初始化和音符的对应关系如表 11.3 所示。

表 11.3 ATmega16 定时器 T/C1 初始化值频率关系

频 率	初 始 化 值	频 率	初 始 化 值
440Hz	0xFEE4	494Hz	0xFF03
554Hz	0xFF1F	587Hz	0xFF2C
659Hz	0xFF43	740Hz	0xFF58
831Hz	0xFF6A	880Hz	0xFF72

一段音乐除了和音符有关系，也和节拍有关系，即 ATmega16 单片机驱动发声器件发出乐声的长度，可以使用延时来实现，表 11.4 则是各个节拍对应的单片机延时长度的关系。

表 11.4 单片机延时和节拍的关系

节拍（1/4 节拍标准）	延 时 长 度	节拍（1/8 节拍标准）	延 时 长 度
4/4	125ms	4/4	62ms
3/4	187ms	3/4	94ms
2/4	250ms	2/4	125ms



注意

在用户使用简易电子琴进行音乐弹奏时，其节拍是由用户自行控制的，而在使用简易电子琴播放实现设置好的音乐时，则需要单片机对节拍进行相应的控制。

11.2.3 硬件系统设计

简易电子琴应用系统的硬件设计重点是合理地划分 ATmega16 单片机的 I/O 引脚，用于驱动不同的外围器件。

1. 硬件系统模块划分

简易电子琴的硬件模块如图 11.10 所示，由 ATmega16 单片机、演奏和播放键、演奏指示灯和发声部件构成，其各部分详细说明如下。

- ATmega16 单片机：这是简易电子琴应用系统的核心控制器。
- 播放键：当被用户按下之后，播放单片机内置的音乐。
- 演奏键：当被用户按下之后，发出对应的音符。
- 发声器件：能够根据 ATmega16 单片机的驱动，发出对应的声音。
- 演奏指示灯：用于指示当前的按键状态。

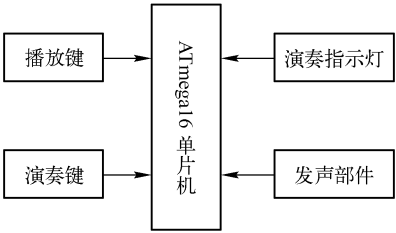


图 11.10 简易电子琴的硬件模块

2. 硬件系统电路

简易电子琴的电路如图 11.11 所示，ATmega16 单片机使用 PB 引脚扩展了 8 个独立按



键，分别对应音调“1”～“#1”，使用 PC0 引脚通过三极管驱动了一个蜂鸣器，8 个发光二极管使用灌电流的方式通过一个 8 位双排阻连接到 ATmega16 单片机的 PD 引脚，用于指示当前的演奏按键工作状态；此外还使用 PA0 引脚扩展了一个按键用于播放预先设置好的音乐。

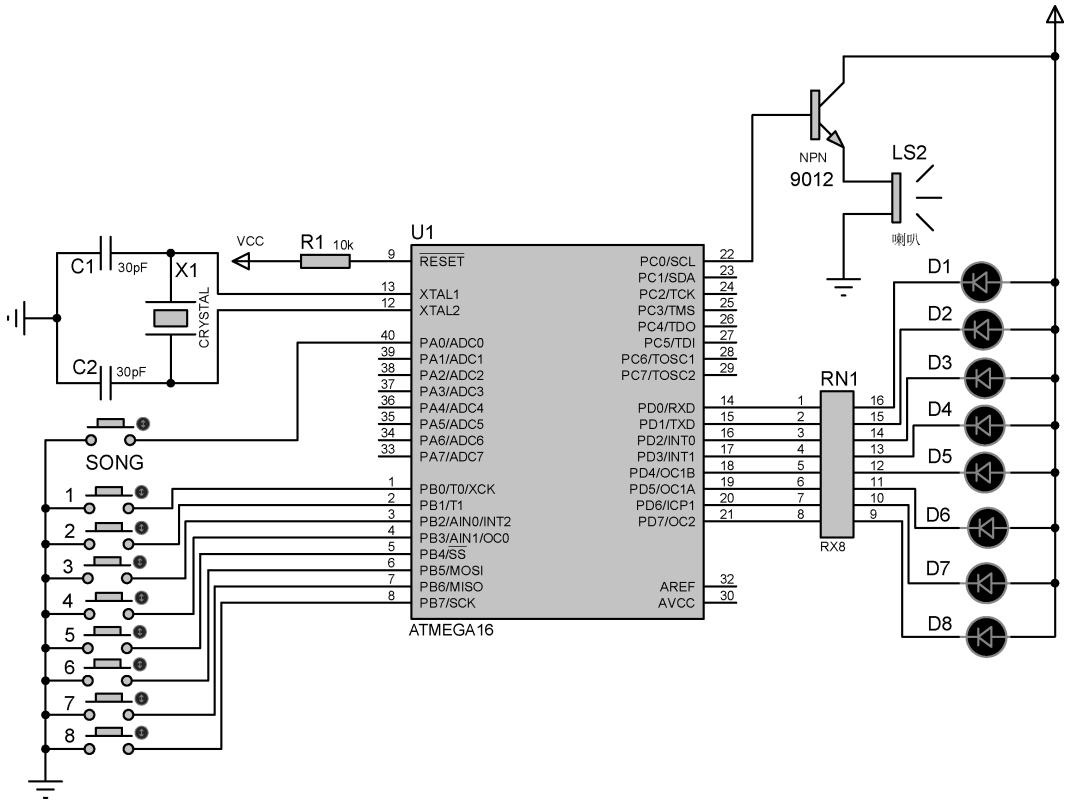


图 11.11 系统的硬件电路

简易电子琴电路涉及的典型器件如表 11.5 所示。

表 11.5 简易电子琴电路涉及的典型器件

器件名称	器件编号	说明
晶体	X1	ATmega16 单片机的振荡源
ATmega16 单片机	U1	ATmega16 单片机，系统的核心控制器件
电容	C1、C2	滤波，储能器件
电阻	R1、R2	限流，上拉
三极管	9012	用于驱动蜂鸣器
独立按键	1～8、SONG	演奏和播放按键
发光二极管	D1～D8	发光器件
蜂鸣器	LS2	发声器件
排阻	RN1	8 位双排阻

### 3. 硬件系统模块介绍——蜂鸣器

按照工作原理，蜂鸣器可以分为压电式蜂鸣器和电磁式蜂鸣器，前者又称为有源蜂鸣器，后者又称为无源蜂鸣器。



#### 注意

有源蜂鸣器和无源蜂鸣器中的“源”指的不是电源，而是振荡源，其最大区别是前者只需要在蜂鸣器两端加上固定的电压差则可激励蜂鸣器发声，而后者必须加上相应频率振荡信号方可。

压电式蜂鸣器（有源蜂鸣器）主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成，多谐振荡器由晶体管或集成电路构成，当接通电源后，多谐振荡器起振，输出 1.5 ~ 2.5kHz 的音频信号，阻抗匹配器推动压电蜂鸣片发声。压电蜂鸣片由锆钛酸铅或铌镁酸铅压电陶瓷材料制成。在陶瓷片的两面镀上银电极，经极化和老化处理后，再与黄铜片或不锈钢片粘在一起。

电磁式蜂鸣器（无源蜂鸣器）由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。接通电源后，振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场。振动膜片在电磁线圈和磁铁的相互作用下，周期性地振动发声。

Proteus 中常用的蜂鸣器模型（SPEAKER）如图 11.12 所示，位于 Speakers & Sounders 库，如图 11.13 所示（图 11.11 则是选择如图 11.13 中的 SOUNDER）。



图 11.12 Proteus 中的继电器模型

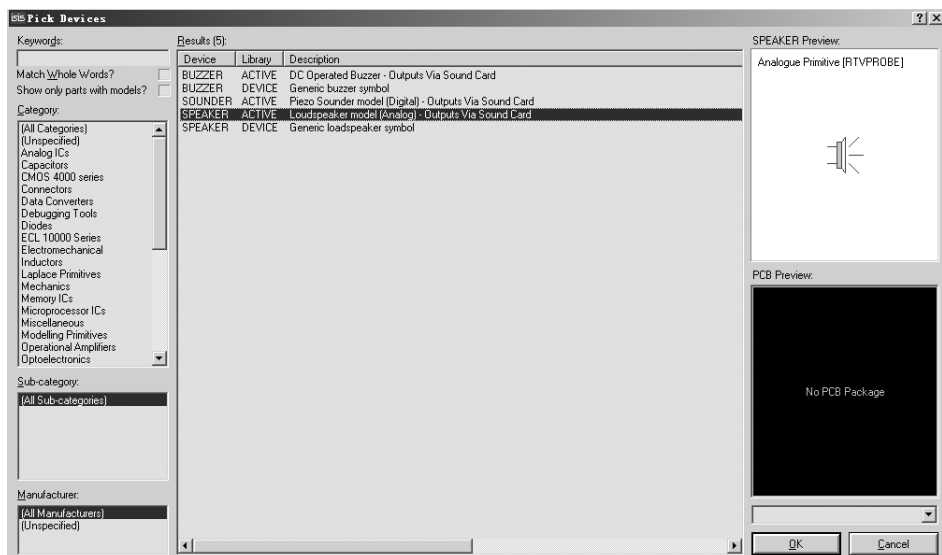


图 11.13 Proteus 中的蜂鸣器

在蜂鸣器上双击，可以弹出如图 11.14 所示的属性设置对话框，其中涉及的主要属性说明如下。

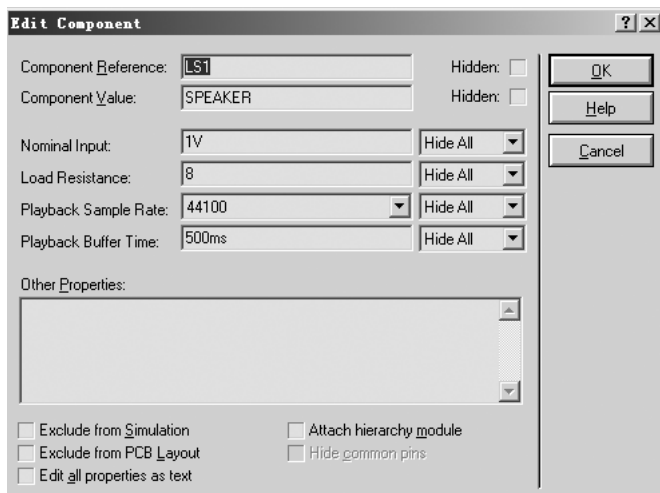


图 11.14 蜂鸣器属性设置对话框

- Nominal Input: 工作电压。
- Load Resistance: 负载电阻。
- Playback Sample Rate: 回放采样率。
- Playback Buffer Time: 回放缓冲时间。

#### 4. 硬件系统模块介绍——三极管

三极管是一种用电流来控制电流的半导体器件，是 ATmega16 单片机应用系统中最常用的功率驱动器件，其作用是把微弱信号放大成幅值较大的电信号，也常常用作无触点开关（如用作多位数码管的选择控制器件）。

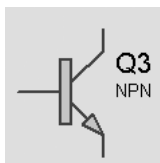


图 11.15 三极管的引脚封装

三极管可以按材料分为锗管和硅管，而每一种按照电流结构又有 NPN 和 PNP 两种形式，但使用最多的是硅 NPN 管和锗 PNP 管两种。

三极管有多种型号，但是其都有 3 个引脚，分别为发射极（Emitter/E）、基极（Base/B）和集电极（Collector/C），如图 11.15 所示。

可以把三极管看作一个电子开关，其中基极是电子开关的控制端，当基极输出高电平时，三极管导通，在被控物体两端形成电压差；当控制端输出低电平时，被控物体两端的电压差消失。



#### 注意

控制端上的电阻必须选取合适，因为较小的电流将不足以使三极管导通。

Proteus 中的三极管位于 Transistors 库中，提供了大量各种特定的三极管，包括 Bipolar、IGBT、JFET 等，如图 11.16 所示。



#### 注意

在实际的 Proteus 的模拟中，通常使用的是 Modelling Primitives 的 Analog 中的 PNP 或者 NPN 模型。

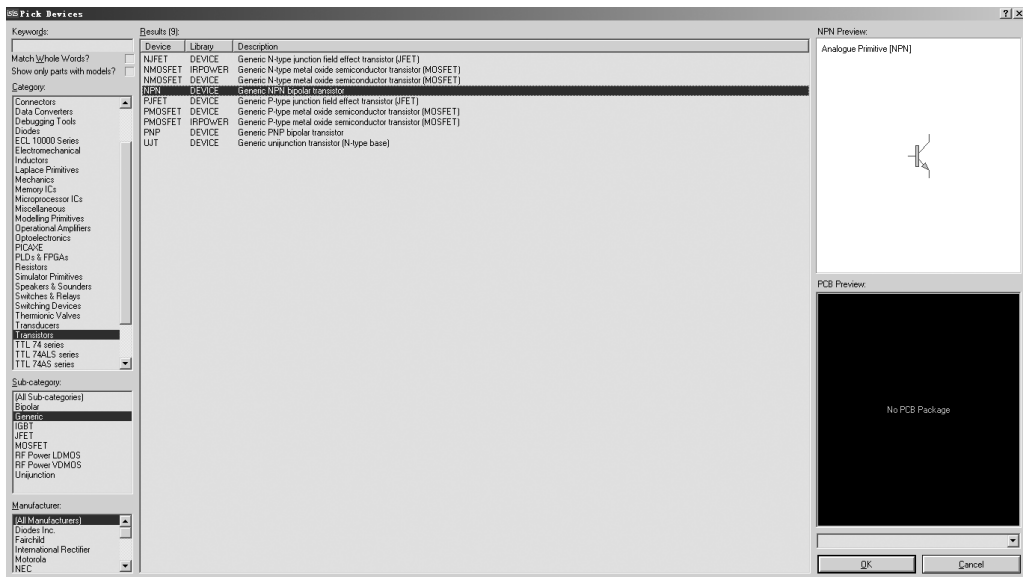


图 11.16 Proteus 中的三极管

在三极管上双击可以弹出如图 11.17 所示的属性设置对话框，通常使用默认值即可。

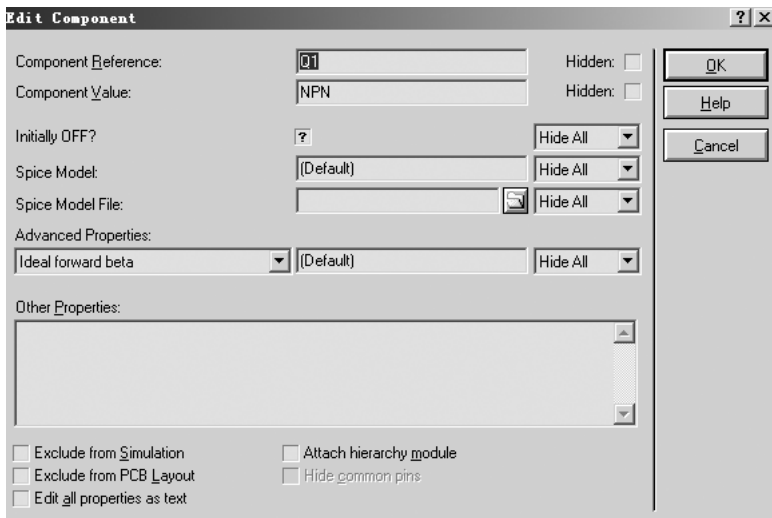


图 11.17 三极管属性设置对话框

### 11.2.4 软件系统设计

简易电子琴软件的设计重点是如何使用定时计数器产生对应的频率波形来驱动蜂鸣器发声，在设计中可以将频率对应的时间常数放在一个数组中，在需要使用时查找输出即可。

#### 1. 软件系统流程

简易电子琴应用系统的软件流程如图 11.18 所示。

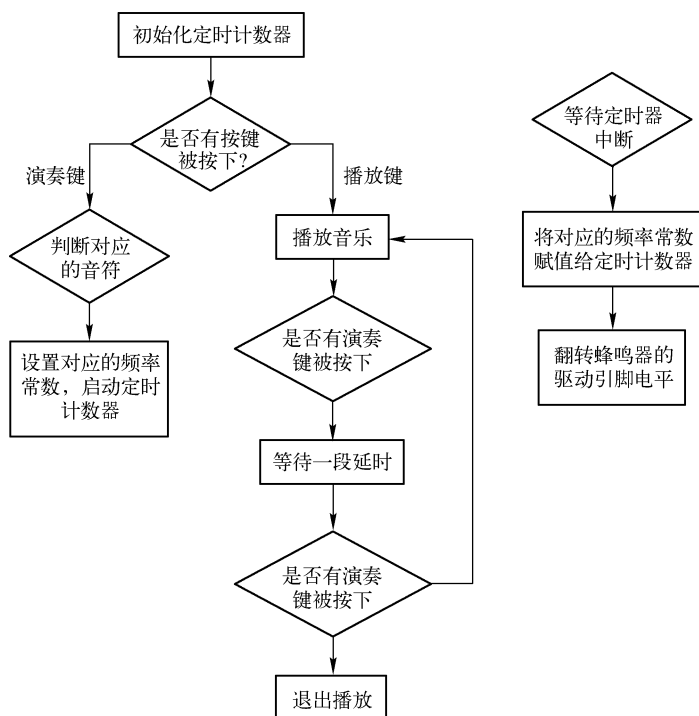


图 11.18 简易电子琴应用系统的软件流程

## 2. 软件系统的应用代码

简易电子琴的应用代码如例 11.2 所示。

应用代码使用了 `freq[ ][2]` 二维数组来存放不同的音符对应的定时计数器初始化值，然后使用 `MUSIC` 数组存放了一首音乐对应的音符数据，以供播放函数 `PlaySong` 调用。在主循环中通过对按键状态的判断来进行不同的处理。

### 【例 11.2】 简易电子琴的应用代码

```

#include <iom16v.h>
#include <macros.h>

unsigned char High, Low; //定时器预装值的高 8 位和低 8 位
unsigned char freq[ ][2] = {
    0xFE, 0xE4, //00440Hz 1
    0xFF, 0x03, //00494Hz 2
    0xFF, 0x1F, //00554Hz 3
    0xFF, 0x2C, //00587Hz 4
    0xFF, 0x43, //00659Hz 5
    0xFF, 0x58, //00740Hz 6
    0xFF, 0x6A, //00831Hz 7
    0xFF, 0x72, //00880Hz 1

```



```

};

unsigned char Time;
unsigned char YINFU[9][1] = { " ", { 1 }, { 2 }, { 3 }, { 4 }, { 5 }, { 6 }, { 7 }, { 8 } };
//《世上只有妈妈好》数据表
unsigned char MUSIC[ ] = { 6,2,3, 5,2,1, 3,2,2, 5,2,2, 1,3,2, 6,2,1, 5,2,1,
                           6,2,4, 3,2,2, 5,2,1, 6,2,1, 5,2,2, 3,2,2, 1,2,1,
                           6,1,1, 5,2,1, 3,2,1, 2,2,4, 2,2,3, 3,2,1, 5,2,2,
                           5,2,1, 6,2,1, 3,2,2, 2,2,2, 1,2,4, 5,2,3, 3,2,1,
                           2,2,1, 1,2,1, 6,1,1, 1,2,1, 5,1,6, 0,0,0
                           };

//音阶频率表, 高 8 位
unsigned char FREQH[ ] = {
    0xF2,0xF3,0xF5,0xF5,0xF6,0xF7,0xF8,
    0xF9,0xF9,0xFA,0xFA,0xFB,0xFB,0xFC,0xFC, //1,2,3,4,5,6,7,8,i
    0xFC,0xFD,0xFD,0xFD,0xFD,0xFE,
    0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0xFF,
    };

//音阶频率表, 低 8 位
unsigned char FREQL[ ] = {
    0x42,0xC1,0x17,0xB6,0xD0,0xD1,0xB6,
    0x21,0xE1,0x8C,0xD8,0x68,0xE9,0x5B,0x8F, //1,2,3,4,5,6,7,8,i
    0xEE,0x44, 0x6B,0xB4,0xF4,0x2D,
    0x47,0x77,0xA2,0xB6,0xDA,0xFA,0x16,
    };

//μs 延时函数
void delay_us(unsigned int n)          //8 × 0.125 = 1μs
{
    int i,j;
    for(j=0;j<8;j++)
    {
        for(i=0;i<n;i++)
            NOP();
    }
}

//ms 延时函数
void delay_ms(unsigned int i)
{

```

```
while(i -- )
{
    unsigned int j;
    for(j = 1 ;j <= 1332 ;j ++ ) ;
}
}
```

//端口初始化函数

```
void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0x00;
    PORTB = 0x00;
    DDRB = 0x00;
    PORTC = 0x00;
    DDRC  = 0x01;
    PORTD = 0x00;
    DDRD  = 0xFF;
}
```

//T/C1 初始化函数, 64 分频

```
void timer1_init( void)
{
    TCCR1B = 0x00;
    TCNT1H = 0xFE;
    TCNT1L = 0xE4;
    OCR1AH = 0x00;
    OCR1AL = 0x7D;
    OCR1BH = 0x00;
    OCR1BL = 0x7D;
    ICR1H = 0x00;
    ICR1L = 0x7D;
    TCCR1A = 0x00;
    TCCR1B = 0x03;
}
```

//设置 TCNT 初始化值

//T/C1 中断溢出处理函数

```
#pragma interrupt_handler timer1_ovf_isr:iv_TIM1_OVF
void timer1_ovf_isr( void)
{
}
```





```

    TCNT1H = High;
    TCNT1L = Low;                //将高位和低位装入定时器
    PORTC^ = BIT(0);            //蜂鸣器翻转
}

/* -----
                               节拍延时函数
    各调 1/4 节拍时间:
    调 4/4   125ms
    调 2/4   250ms
    调 3/4   187ms
    ----- */
void delayTips(unsigned char t)
{
    unsigned char i;
    for(i = 0; i < t; i++)
    {
        delay_ms(250);
    }
    TCCR1B = 0x00;                //关闭
}
//播放音乐的函数
void PlaySong(void)
{
    TCNT1H = High;
    TCNT1L = Low;                //装载初始化值
    TCCR1B = 0x03;                //定时器开启
    delayTips(Time);              //延时所需要的节拍
}

//初始化 ATmega16
void init_devices(void)
{
    CLI();
    port_init();
    timer1_init();

    MCUCR = 0x00;
    GICR  = 0x00;
    TIMSK = 0x04;
    SEI();

```

```

}

//主函数
void main( void)
{
    unsigned char num,k,i;
    unsigned char KeyPort = 0x00;
    unsigned char KeyPlaySong = 0x00;
    init_devices( );
    PORTC = 0x00; //蜂鸣器无输出
    while( 1)
    {
        PORTB = 0xFF;
        KeyPort = PINB;
        PORTD = KeyPort;
        switch( KeyPort)
        {
            case 0xfe: num = 1; break;
            case 0xfd: num = 2; break;
            case 0xfb: num = 3; break;
            case 0xf7: num = 4; break;
            case 0xef: num = 5; break;
            case 0xdf: num = 6; break;
            case 0xbf: num = 7; break;
            case 0x7f: num = 8; break; //分别对应不同的音调
            default: num = 0; break;
        }
        if( num == 0)
        {
            TCCR1B = 0x00; //定时器关闭
            PORTC = 0x00; //在未按键时，喇叭低电平，防止长期高电平损坏喇叭
        }
        else
        {
            High = freq[ num - 1 ][ 0 ];
            Low = freq[ num - 1 ][ 1 ];
            TCCR1B = 0x03; //定时器开启
        }
        PORTA = 0xFF;
        KeyPlaySong = PINA;
        if( KeyPlaySong == 0xFE) //如果 play 按键被按下
    }
}

```



```

delay_ms(10); //延时 10ms
PORTA = 0xFF;
KeyPlaySong = PINA;
if( KeyPlaySong == 0xFE) //如果 play 按键被按下
{
    i = 0;
    while( i < 100)
    {
        k = MUSIC[ i ] + 7 * MUSIC[ i + 1 ] - 1; //去音符振荡频率所需数据
        High = FREQH[ k ];
        Low = FREQL[ k ];
        Time = MUSIC[ i + 2 ]; //节拍时长
        i = i + 3;
        PORTB = 0xFF;
        if( PINB != 0xFF) //长按任意 8 音键退出播放
        {
            delay_ms(10);
            if( PINB != 0xFF)
            {
                i = 101;
            }
        }
        PlaySong();
    }
    TCCR1B = 0x03; //定时器开启
}
}
}
}

```

### 11.2.5 应用系统的仿真和总结

在 Proteus 中绘制如图 11.11 所示的电路，其中所涉及的典型器件如表 11.6 所示。

表 11.6 Proteus 电路器件列表

器件名称	库	子库	说明
ATmega16	Microprocessor ICs	AVR Family	ATmega16
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CAP - ELEC	Capacitors	Generic	极性电容

续表

器件名称	库	子 库	说 明
CRYSTAL	Miscellaneous	—	晶体
NPN	Modelling Primitives	Analog	NPN 三极管
SPEAKER	Speakers & Sounders	—	蜂鸣器
BUTTON	Switches & Relays	Switches	独立按键
LED - RED	Optoelectronics	LEDs	发光二极管（黄色）

点击运行，分别按下对应演奏按键，可以听到对应的音符并且看到对应发光二极管被点亮，如果按下了播放按键，则可以听到音乐播放，在播放音乐时如果长按任意一个播放键，则可以退出播放状态，如图 11.19 所示。

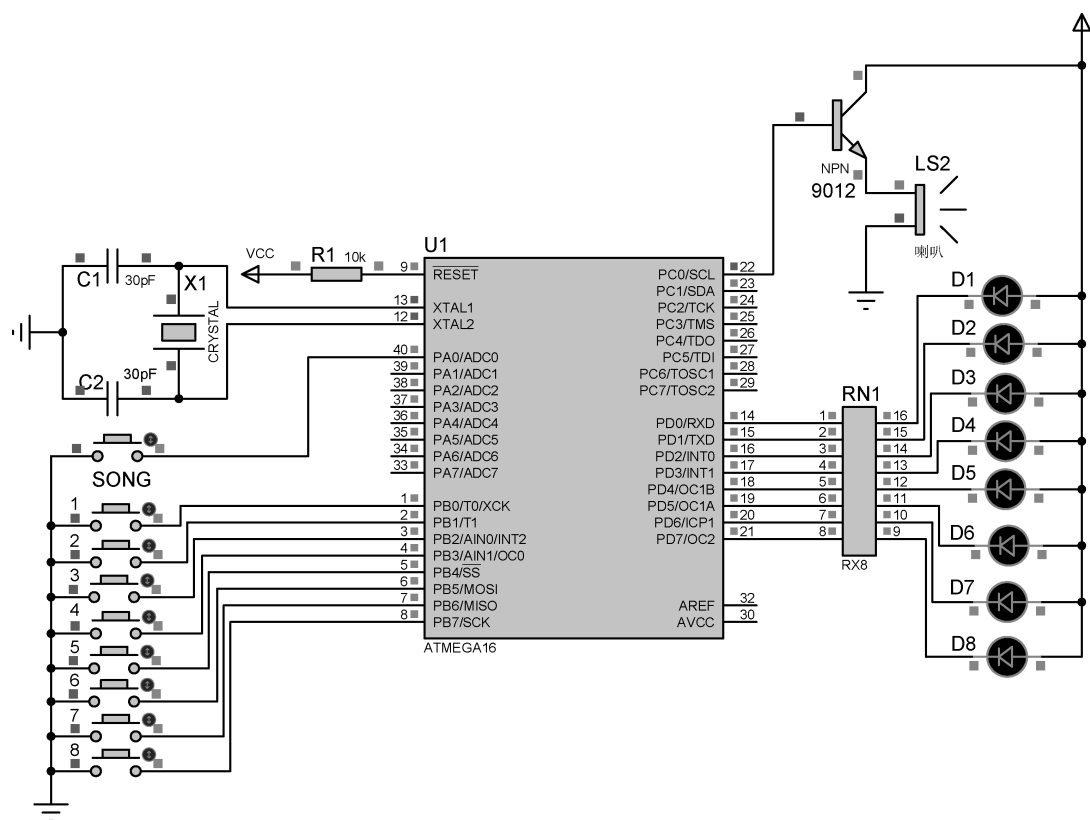


图 11.19 简易电子琴的 Proteus 仿真



## 总结

在实际应用中，可以使用不同的乐声来表示 ATmega16 单片机应用系统的不同状态，如报警状态、空闲状态等；还可以使用同样的原理制作电子门铃等应用系统。



## 11.3 商场灯光控制

商场、大型购物中心等场合，在夜间或者人比较少时也需要开启全部灯光，这是一种浪费能源的行为，但是为了“效果”又不得不为之，商场灯光控制系统可以比较好地协调“效果”和能源。

### 11.3.1 应用系统背景

大多数灯光的亮度和能耗与加在灯两端的电压值有关，当电压值较低时，其能耗较低，亮度也较低；对于人眼来说，亮度由流明数决定，但是肉眼并不能分辨出其比较细微的差别，所以商场灯光节能控制系统可以利用在人数较少时采用较低电压（如 210V、200V 等）对灯泡进行供电，以达到节省能源的目的。

### 11.3.2 设计思路

#### 1. 系统工作流程

商场灯光控制系统的工作流程如图 11.20 所示。

#### 2. 系统的需求分析与设计

设计商场灯光控制系统，需要考虑如下几个方面的内容。

- 如何获得当前的时间信息。
- 如何切换当前的供电电压。
- 用什么模块来显示当前的时间信息。
- 需要设计合适的单片机软件。

### 11.3.3 硬件系统设计

#### 1. 硬件系统模块划分

商场灯光控制系统的硬件模块如图 11.21 所示，由 ATmega16 单片机、时钟模块、继电器控制模块和显示模块组成，其各个部分详细说明如下。

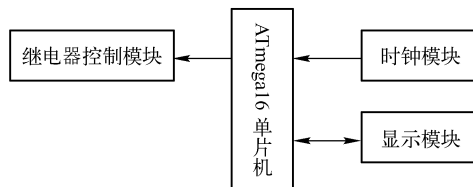


图 11.21 商场灯光控制系统的硬件模块

- ATmega16 单片机：商场灯光控制系统的核心控制器。
- 时钟模块：用于提供相应的时间信息。
- 继电器控制模块：对相应的电压进行切换。

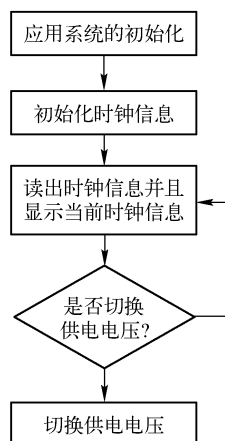


图 11.20 商场灯光控制系统的工作流程



- 显示模块：显示当前的时钟信息。

## 2. 硬件系统电路

商场灯光控制系统的硬件电路如图 11.22 所示，ATmega16 单片机使用 PA 引脚和 PC 的部分引脚驱动了一片 1602 液晶模块，用于显示相应的时间信息，使用 PB5 ~ PB7 引脚模拟 SPI 总线与时钟芯片 DS1302 通信，同时使用 PD6、PD7 作为继电器控制引脚。

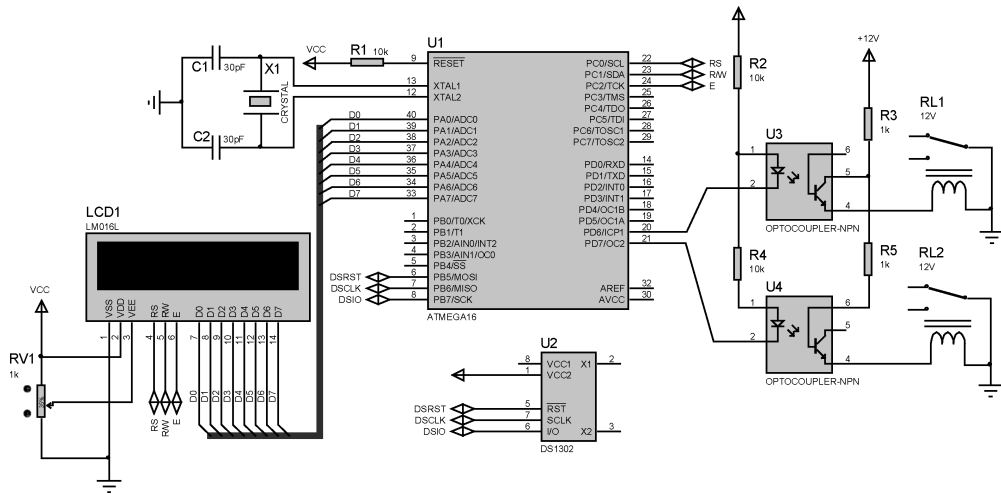


图 11.22 商场灯光控制系统的硬件电路

商场灯光控制系统涉及的典型器件如表 11.7 所示。

表 11.7 商场灯光控制系统涉及的典型器件

器件名称	器件编号	说明
晶体	X1	ATmega16 单片机的振荡源
ATmega16 单片机	U1	ATmega16 单片机，系统的核心控制器件
电容	C1、C2	滤波，储能器件
电阻	R1、R2、R3 等	上拉
滑动变阻器	RV1	调节液晶显示器
DS1302	U2	时钟芯片
LM016L	LCD1	1602 液晶模块
继电器	U3、U4	控制器件

## 3. 硬件系统模块介绍——1602 液晶模块

数码管虽然能显示数字和一些简单的字符，但它还是有一些局限性，如不能显示“Hello World!”这样的语句，此时可以使用液晶模块（如 1602），图 11.23 为 1602 液晶的引脚示意图，其详细说明如下。

- VSS：电源地信号引脚。
- VDD：电源信号引脚。



图 11.23 1602 的引脚

- VEE：液晶对比度调节引脚，接 0 ~ 5V 以调节液晶的显示对比度。
- RS：寄存器选择引脚，当该引脚为高电平时选择的是数据寄存器，为低电平时选择的是指令寄存器。
- RW：读、写操作选择引脚，当该引脚为高电平时选择为读操作，反之为写操作。
- E：使能信号引脚，当该引脚为低电平时，数据被写入 1602；当该引脚为高电平时，可以对 1602 进行数据读操作。
- D0 ~ D7：数据总线引脚。
- LEDA：背光电源引脚。
- LEDK：背光电源地引脚。

1602 液晶支持一系列指令，包括清屏命令、归零命令等，其详细说明如下。

- 清屏指令：用于清除 DDRAM 和 AC 的数值，将屏幕的显示清空，如表 11.8 所示。

表 11.8 清屏指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1

- 归零指令：将屏幕的光标回归原点，如表 11.9 所示。

表 11.9 归零指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	1	*

- 输入方式选择指令；用于设置光标和画面移动方式。其中：I/D = 1，数据读、写操作后，AC 自动加 1；I/D = 0，数据读、写操作后，AC 自动减 1；S = 1，数据读、写操作，画面平移；S = 0，数据读、写操作，画面保持不变，如表 11.10 所示。

表 11.10 输入方式选择指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	1	I/D	S

- 显示开关控制指令：用于设置显示、光标及闪烁开、关。其中：D 表示显示开关，D = 1 为开，D = 0 为关；C 表示光标开关，C = 1 为开，C = 0 为关；B 表示闪烁开关，



B = 1 为开, B = 0 为关, 如表 11.11 所示。

表 11.11 显示开关控制指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	1	D	C	B

- 光标和画面移动指令：用于在不影响 DDRAM 的情况下使光标、画面移动。其中：S/C = 1, 画面平移一个字符位；S/C = 0, 光标平移一个字符位；R/L = 1, 右移；R/L = 0, 左移, 如表 11.12 所示。

表 11.12 光标和画面移动指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	1	S/C	R/L	*	*

- 功能设置指令：用于设置工作方式（初始化指令）。其中：DL = 1, 8 位数据接口；DL = 0, 4 位数据接口；N = 1, 两行显示；N = 0, 一行显示；F = 1, 5 × 10 点阵字符；F = 0, 5 × 7 点阵字符, 如表 11.13 所示。

表 11.13 功能设置指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	DL	N	F	*	*

- CGRAM 设置指令：用于设置 CGRAM 地址, A5 ~ A0 = 0x00 ~ 0x3F, 如表 11.14 所示。

表 11.14 CGRAM 设置指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	A5	A4	A3	A2	A1	A0

- DDRAM 设置指令：用于设置 DDRAM 地址。N = 0, 一行显示, A6 ~ A0 = 0 ~ 4FH；N = 1, 两行显示, 首行 A6 ~ A0 = 00H ~ 2FH, 次行 A6 ~ A0 = 40H ~ 64FH, 如表 11.15 所示。

表 11.15 DDRAM 设置指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	A6	A5	A4	A3	A2	A1	A0

- 读 BF 和 AC 指令：BF = 1 表示忙；BF = 0 表示准备好。此时, AC 值意义为最近一次地址设置（CGRAM 或 DDRAM）定义, 如表 11.16 所示。

表 11.16 读 BF 和 AC 指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0



- 写数据指令：用于将地址码写入 DDRAM，以使 LCD 显示出相应的图形或将用户自创的图形存入 CGRAM 内，如表 11.17 所示。

表 11.17 写数据指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
1	0	数据							

- 读数据指令：根据当前设置的地，从 DDRAM 或 CGRAM 数据读出，如表 11.18 所示。

表 11.18 读数据指令

RS	RW	D7	D6	D5	D4	D3	D2	D1	D0
1	1	数据							

Proteus 中的 1602 液晶模块名称为 LM016L，位于 Optoelectronics 库的 Alphanumeric LCDs 子分类库中，该分类库还提供了一些其他字符液晶的模型，如图 11.24 所示。

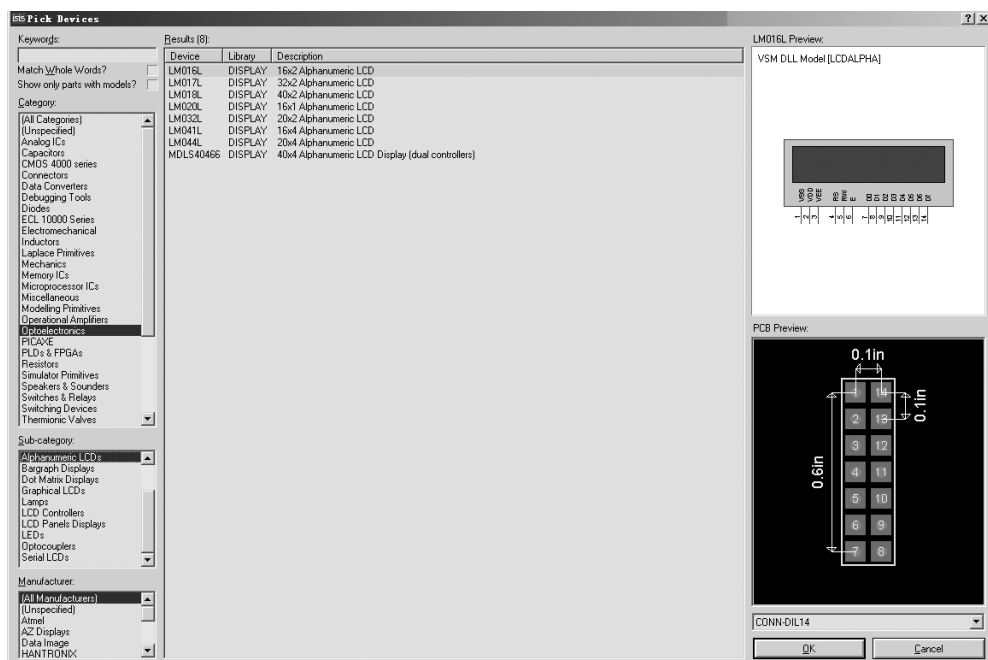


图 11.24 Proteus 中 1602 库

在 1602 上双击，可以弹出如图 11.25 所示的属性设置对话框，其中涉及的主要参数说明如下。

- Clock Frequency：1602 的内部时钟频率，提高这个频率可以加快液晶模块的响应速度，通常来说使用默认值即可。
- Row1：1602 的第 1 行内部地址，默认为 0x80 ~ 0x8F，通常采用默认值即可。

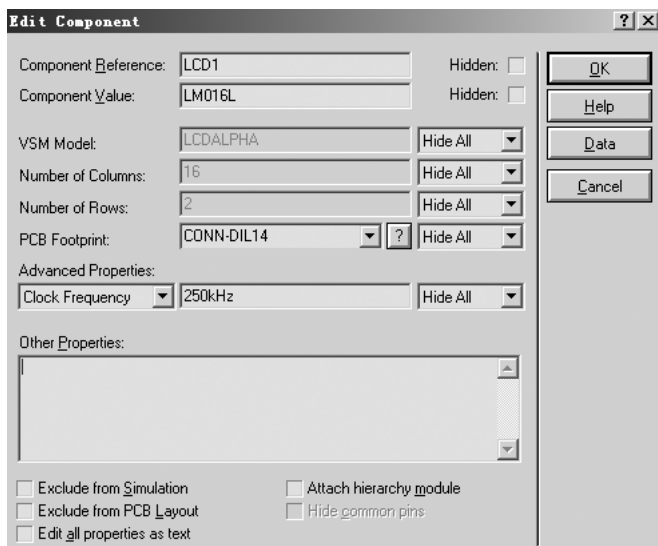


图 11.25 1602 的属性设置对话框

- Row2: 1602 的第 2 行内部地址，默认为 0xC0 ~ 0xCF，通常采用默认值即可。

#### 4. 硬件系统模块介绍——DS1302 时钟模块

在 ATmega16 的应用系统中，常常需要获取当前的时间信息，如定时数据采集或者报警等，此时可以使用时钟芯片来完成相应的功能。

DS1302 是 DALLAS 公司生产的另外一款使用串行接口的时钟日历芯片，其主要特点如下。

- 使用 SPI 三线接口和 ATmega16 通信。
- 内置 31 字节 RAM。
- 提供秒、分、时、日、星期、月、年数据，其中月计数 30 与 31 天可以自动调整，且具有闰年补偿功能。
- 提供 2.5 ~ 5.5V 宽电压工作，采用双电源供电（主电源和备用电源），并且可设置备用电源充电方式。

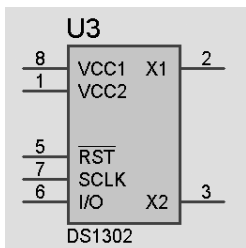


图 11.26 DS1302 的引脚

图 11.26 是 DS1302 的引脚，其详细说明如下。

- VCC1: 主电源输入引脚。
- VCC2: 备份电源输入引脚，当  $VCC2 > VCC1 + 0.2V$  时，由 VCC2 向 DS1302 供电，当  $VCC2 < VCC1$  时，由 VCC1 向 DS1302 供电。
- SCLK: 串行时钟引脚。
- I/O: 数据引脚。
- RST: 功能控制引脚，高有效。
- X1、X2: 外部晶体信号输入引脚。

DS1302 的寄存器可以分为时间寄存器、控制寄存器、突发传输寄存器、充电寄存器等，



表 11.19 是 DS1302 寄存器的地址分布。

表 11.19 DS1302 寄存器的地址分布

读寄存器	写寄存器	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	范 围
0x81	0x80	CH	10 秒			秒				00～59
0x83	0x82		10 分			分				00～59
0x85	0x84	12/24	0	10	小时	小时				1～12/0～23
				AM/PM						
0x87	0x86	0	0	10 日		日				1～31
0x89	0x88	0	0	0	10 月	月				1～12
0x8B	0x8A	0	0	0	0	0	周日			1～7
0x8D	0x8C	10 年				年				00～99
0x8F	0x8E	WP	0	0	0	0	0	0	0	—

如表 11.19 所示，小时寄存器（0x85、0x84）的 BIT7 用于定义 DS1302 是运行于 12 小时模式还是 24 小时模式，当为该位为“1”时，选择 12 小时模式，在 12 小时模式下，BIT5 用于标志上午还是下午，当 BIT5 为“1”时，表示 PM，为“0”时表示 AM。在 24 小时模式时，BIT5 是第二个 10 小时位。

秒寄存器（0x81、0x80）的 BIT7 定义为时钟暂停标志位（CH），当该位置为“1”时，时钟振荡器停止，DS1302 处于低功耗状态；当该位置为 0 时，时钟开始运行。

控制寄存器（0x8F、0x8E）的 BIT7 是写保护位（WP），其他 7 位均置为 0，在任何对时钟和 RAM 写操作之前，WP 位必须为 0。当 WP 位为 1 时，写保护位防止对任意一个寄存器的写操作。

DS1302 的控制字用于在 ATmega16 和 DS1302 进行通信时选择对应的寄存器以及决定操作内容，其结构如表 11.20 所示，详细说明如下。

表 11.20 DS1302 的控制字

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1	RAM	A4	A3	A2	A1	A0	RD
	/CK						/WR

- BIT7：必须是“1”，如果该位为“0”，则不能把数据写入到 DS1302 中。
- BIT6：为“0”表示操作日历时钟寄存器，为“1”表示操作 RAM 空间。
- BIT5 ~ BIT1：寄存器或者内部 RAM 地址。
- BIT0：读写指示位，为“0”表示进行写操作，为“1”表示进行读操作。

DS1302 的控制字总是从最低位开始输出的。在控制字指令输入后的下一个 SCLK 时钟的上升沿时，数据被写入 DS1302，数据输入从 BIT0 开始。同样，在紧跟 8 位的控制字指令后的下一个 SCLK 脉冲的下降沿，读出 DS1302 的数据，读出的数据也是从最低位到最高位。

Proteus 中的 DS1302 位于 Microprocessor ICs 库的 Peripherals 子分类库中，提供了多种

ATmega16 的外围接口器件，如图 11.27 所示。

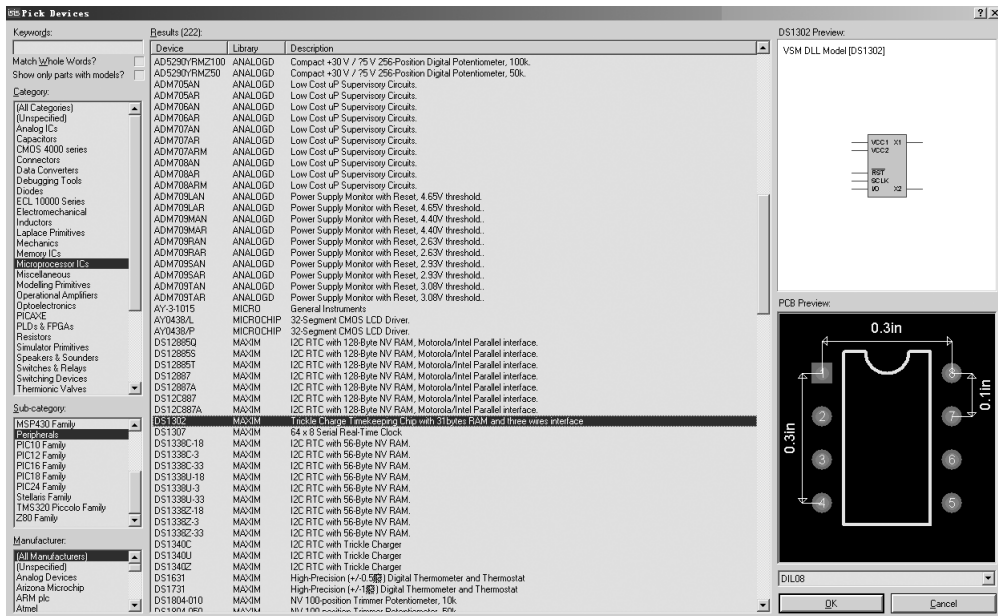


图 11.27 Proteus 中的 DS1302 库

在 DS1302 上双击，可以弹出如图 11.28 所示的属性设置对话框，其中除了元件电路图标号外没有需要用户修改的属性。

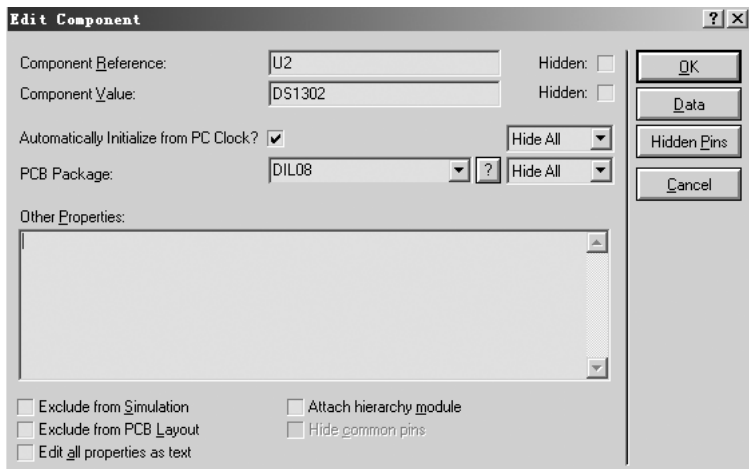


图 11.28 DS1302 的属性设置对话框



注意

在 Proteus 电路仿真中，DS1302 使用了 PC 的系统时钟作为其内部默认时钟数据。

## 5. 硬件系统模块介绍——继电器

在某些 ATmega16 的应用系统中，需要使用 I/O 引脚来控制一些大电流设备的启动或者停止，如电磁铁，此时就需要使用继电器作为中间介质，使用单片机的 I/O 引脚来控制继电器的通断，然后再使用继电器来控制这些设备的启动或者停止。

继电器是一种电子控制器件，它由控制系统（又称输入回路）和被控制系统（又称输出回路）组成，通常应用于自动控制电路中，其实质是用较小电流去控制较大电流的一种“自动开关”，在应用系统中起着自动调节、安全保护、转换电路等作用，在 ATmega16 系统中常常用于通断控制。

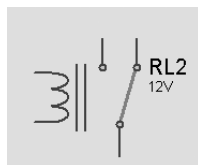


图 11.29 Proteus 中的继电器模型

Proteus 中常用的继电器模型如图 11.29 所示，其中左侧的两个引脚分别为继电器引脚的两个控制端，右侧的三个引脚是被控制端。当控制引脚两端没有电压差时，右侧的公共引脚和其中一个引脚导通；而当控制引脚两端有电压差时，右侧的公共引脚和其中另外一个引脚导通。

Proteus 中的继电器模型（RELAY）模型位于 Switches & Relays 库的 Relays（Generic）子类库，如图 11.30 所示。

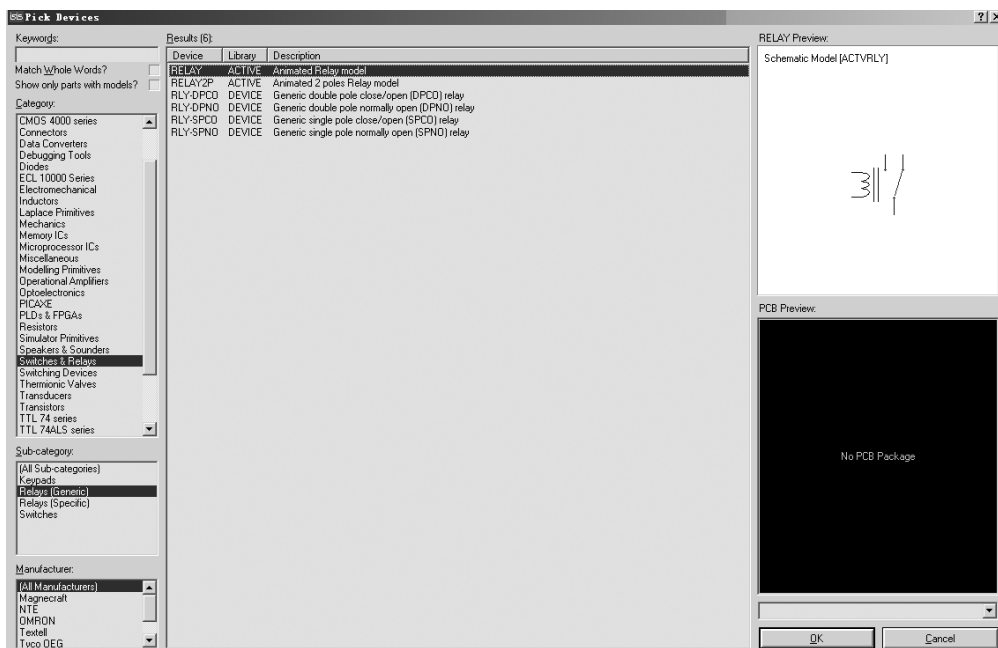


图 11.30 Proteus 中的继电器模型

在继电器上双击，可以弹出如图 11.31 所示的属性设置对话框，其中涉及的主要属性说明如下。

- Coil Resistance：线圈电阻。
- Advanced Properties：高级设置，通常使用默认值即可。

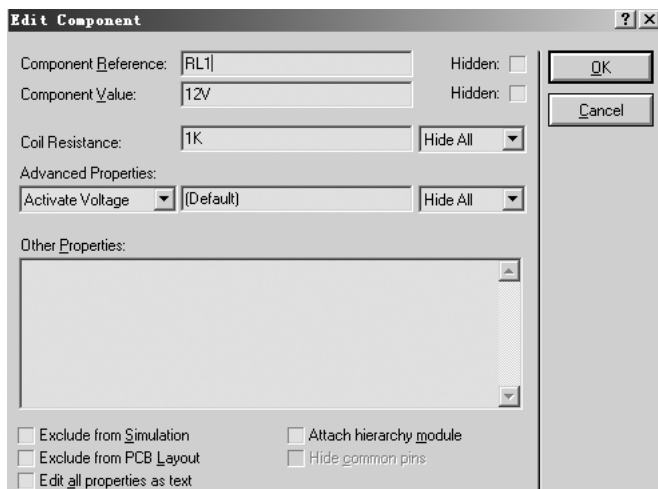


图 11.31 继电器的属性设置对话框



注意

在 Switches & Relays 的 Relays (Specific) 子类库中，还有大量其他的指定类型的继电器，如 G2R、T73S5D 等。

### 11.3.4 软件系统设计

#### 1. 软件系统流程

商场灯光控制系统的软件包括 DS1302 驱动模块和 1602 液晶驱动模块两部分，其流程如图 11.32 所示。

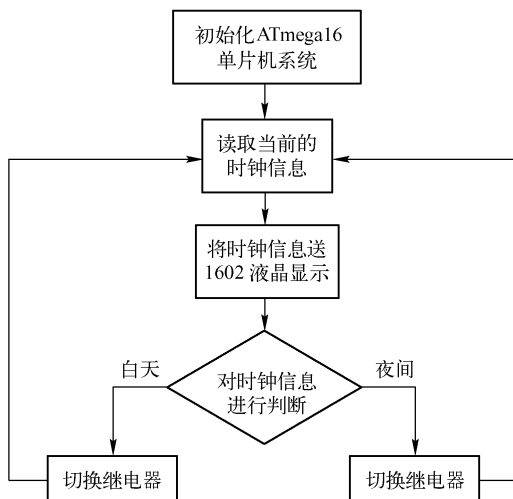


图 11.32 商场灯光控制系统的软件流程



## 2. DS1302 驱动函数模块设计

应用代码使用 ATmega16 单片机的普通 I/O 引脚模拟了 SPI 总线时序和 DS1302 进行通信, 此时需要选择合适延时长度以满足相应时序。DS1302 的驱动函数主要用于对 DS1302 进行相应的操作, 包括如下的操作函数, 其应用代码如例 11.3 所示。

- void ds1302\_init(void): DS1302 初始化函数。
- void ds1302\_write\_byte(unsigned char addr, unsigned char d): 向 DS1302 写入一字节数据。
- unsigned char ds1302\_read\_byte(unsigned char addr): 从 DS1302 读出一字节数据。
- void ds1302\_write\_time(void): 向 DS302 写入时钟数据。
- void ds1302\_read\_time(void): 从 DS302 读出时钟数据。

### 【例 11.3】 DS1302 驱动函数的应用代码

```

/ ***** DS1302 复位引脚 ***** /
#define RST_CLR  PORTB &=~ (1 << PB5)      /* 电平置低 */
#define RST_SET  PORTB |= (1 << PB5)       /* 电平置高 */
#define RST_IN   DDRB &=~ (1 << PB5)       /* 方向输入 */
#define RST_OUT  DDRB |= (1 << PB5)        /* 方向输出 */

/ ***** DS1302 双向数据 ***** /
#define IO_CLR   PORTB &=~ (1 << PB7)      /* 电平置低 */
#define IO_SET   PORTB |= (1 << PB7)       /* 电平置高 */
#define IO_R     PINB & (1 << PB7)         /* 电平读取 */
#define IO_IN    DDRB &=~ (1 << PB7)       /* 方向输入 */
#define IO_OUT   DDRB |= (1 << PB7)        /* 方向输出 */

/ ***** DS1302 时钟信号 ***** /
#define SCK_CLR  PORTB &=~ (1 << PB6)      /* 时钟信号 */
#define SCK_SET  PORTB |= (1 << PB6)       /* 电平置高 */
#define SCK_IN   DDRB &=~ (1 << PB6)       /* 方向输入 */
#define SCK_OUT  DDRB |= (1 << PB6)        /* 方向输出 */

/ ***** DS1302 内部空间定义 ***** /
#define ds1302_sec_add      0x80          //秒数据地址
#define ds1302_min_add      0x82          //分数数据地址
#define ds1302_hr_add       0x84          //时数据地址
#define ds1302_date_add     0x86          //日数据地址
#define ds1302_month_add    0x88          //月数据地址
#define ds1302_day_add      0x8a          //星期数据地址
#define ds1302_year_add     0x8c          //年数据地址
#define ds1302_control_add  0x8e          //控制数据地址

```



```
#define ds1302_charger_add    0x90
#define ds1302_clkburst_add  0xbe

unsigned char time_buf[8] = {0x20,0x09,0x04,0x28,0x17,0x13,0x00,0x02};

//DS1302 初始化函数
void ds1302_init(void)
{
    RST_CLR;                /* RST 引脚置低 */
    SCK_CLR;                /* SCK 引脚置低 */
    RST_OUT;                /* RST 引脚设置为输出 */
    SCK_OUT;                /* SCK 引脚设置为输出 */
}

//向 DS1302 写入一字节数据
void ds1302_write_byte(unsigned char addr, unsigned char d)
{
    unsigned char i;
    RST_SET;                /* 启动 DS1302 总线 */
    //写入目标地址:addr
    IO_OUT;
    addr = addr & 0xFE;      /* 最低位置 0, 寄存器 0 位为 0 时写, 为 1 时读 */
    for (i=0; i<8; i++)
    {
        if (addr & 0x01)
        {
            IO_SET;
        }
        else {
            IO_CLR;
        }
        SCK_SET;            /* 产生时钟 */
        SCK_CLR;
        addr = addr >> 1;
    }
    //写入数据
    IO_OUT;
    for (i=0; i<8; i++)
    {
        if (d & 0x01)
        {
            IO_SET;
        }
    }
}
```





```

        else {
            IO_CLR;
        }

        SCK_SET;           //产生时钟
        SCK_CLR;
        d = d >> 1;
    }

    RST_CLR;               //停止 DS1302 总线
}

//从 DS1302 读出一字节数据
unsigned char ds1302_read_byte(unsigned char addr)
{
    unsigned char i;
    unsigned char temp;
    RST_SET;               /* 启动 DS1302 总线 */
    //写入目标地址:addr
    IO_OUT;
    addr = addr | 0x01;    //最低位置高, 寄存器 0 位为 0 时写, 为 1 时读
    for (i=0; i<8; i++)
    {
        if (addr & 0x01)
        {
            IO_SET;
        }
        else {
            IO_CLR;
        }

        SCK_SET;
        SCK_CLR;
        addr = addr >> 1;
    }

    /* 输出数据:temp */
    IO_IN;
    for (i=0; i<8; i++)
    {
        temp = temp >> 1;
        if (IO_R)
        {
            temp |= 0x80;
        }
    }
}

```



```
        else {
            temp &= 0x7F;
        }

        SCK_SET;
        SCK_CLR;
    }

    RST_CLR;          /* 停止 DS1302 总线 */
    return temp;
}

//向 DS302 写入时钟数据
void ds1302_write_time(void)
{
    ds1302_write_byte(ds1302_control_add,0x00);          //关闭写保护
    ds1302_write_byte( ds1302_sec_add,0x80);             //暂停时钟
    ds1302_write_byte( ds1302_year_add,time_buf[ 1] );   //年
    ds1302_write_byte( ds1302_month_add,time_buf[ 2] );  //月
    ds1302_write_byte( ds1302_date_add,time_buf[ 3] );   //日
    ds1302_write_byte( ds1302_hr_add,time_buf[ 4] );     //时
    ds1302_write_byte( ds1302_min_add,time_buf[ 5] );    //分
    ds1302_write_byte( ds1302_sec_add,time_buf[ 6] );    //秒
    ds1302_write_byte( ds1302_day_add,time_buf[ 7] );    //周
    ds1302_write_byte( ds1302_control_add,0x80);         //打开写保护
}

//从 DS302 读出时钟数据
void ds1302_read_time(void)
{
    time_buf[ 1] = ds1302_read_byte( ds1302_year_add );   //年
    time_buf[ 2] = ds1302_read_byte( ds1302_month_add ); //月
    time_buf[ 3] = ds1302_read_byte( ds1302_date_add );  //日
    time_buf[ 4] = ds1302_read_byte( ds1302_hr_add );    //时
    time_buf[ 5] = ds1302_read_byte( ds1302_min_add );   //分
    time_buf[ 6] = ( ds1302_read_byte( ds1302_sec_add ) )&0x7F; //秒, 屏蔽秒的第 7 位, 避免超出 59
    time_buf[ 7] = ds1302_read_byte( ds1302_day_add );   //周
}
```

### 3. 1602 液晶驱动函数模块设计

1602 液晶驱动函数模块包含用于驱动 1602 液晶模块的相应函数, 包括如下的操作函数, 其应用代码如例 11.4 所示, 应用代码使用了移位操作对 1602 液晶模块进行控制。

- void LCD\_init(void): 显示屏初始化函数。
- void LCD\_en\_write(void): 控制 LCD 写时序。
- void LCD\_clear(void): 清屏函数。



- void Write\_Instruction(unsigned char command): 写指令函数。
- void Write\_Data(unsigned char Wdata): 写数据函数。
- void LCD\_SET\_XY(unsigned char X,unsigned char Y): 字符显示初始地址设置。
- void LCD\_write\_str(unsigned char X,unsigned char Y,unsigned char \*s): 在第 X 行、第 Y 列开始显示, 指针 \*s 所指向的字符串。
- void LCD\_write\_char(unsigned char X,unsigned char Y,unsigned char Wdata): 在第 X 行、第 Y 列开始显示 Wdata 所对应的单个字符。

#### 【例 11.4】 LCD1602 驱动函数的应用代码

```

/ ***** 1602 液晶模块控制引脚定义 ***** /
#define RS_CLR  PORTC &= ~(1 << PC0)
#define RS_SET  PORTC |= (1 << PC0)
#define RW_CLR  PORTC &= ~(1 << PC1)
#define RW_SET  PORTC |= (1 << PC1)
#define EN_CLR  PORTC &= ~(1 << PC2)
#define EN_SET  PORTC |= (1 << PC2)

//控制 LCD 写时序
void LCD_en_write( void)           //EN 端产生一个高电平脉冲, 控制 LCD 写时序
{
    EN_SET;
    delay_us(3);
    EN_CLR;
    delay_us(3);
}

//写指令函数
void Write_Instruction(unsigned char command)
{
    RS_CLR;
    RW_CLR;
    EN_SET;
    PORTA = command;
    LCD_en_write();               //写入指令数据
}

//显示屏初始化函数
void LCD_init( void)
{
    DDRA = 0xFF;                  //I/O 口方向设置
    DDRC |= (1 << PC0) | (1 << PC1) | (1 << PC2);
}

```



```
delay_ms(15);           //上电延时一段时间, 使供电稳定
Write_Instruction(0x38); //8bit 接口, 2 线, 5 × 7 点
delay_ms(5);
Write_Instruction(0x38);
delay_ms(5);
Write_Instruction(0x38);
Write_Instruction(0x08); //关显示, 不显光标, 光标不闪烁
Write_Instruction(0x01); //清屏
delay_ms(5);
Write_Instruction(0x04); //写一字符, 整屏显示不移动
delay_ms(5);

Write_Instruction(0x0C); //开显示, 光标、闪烁都关闭
}

//清屏函数
void LCD_clear( void)
{
    Write_Instruction(0x01);
    delay_ms(2);
}

//写数据函数
void Write_Data(unsigned char Wdata)
{
    RS_SET;
    RW_CLR;
    EN_SET;
    PORTA = Wdata;
    LCD_en_write(); //写入数据
}

//字符显示初始地址设置
void LCD_SET_XY(unsigned char X,unsigned char Y)
{
    unsigned char address;
    if( Y == 0)
        address = 0x80 + X; //Y=0, 表示在第一行显示, 地址基数为 0x80
    else
        address = 0xc0 + X; //Y≠0 时, 表时在第二行显示, 地址基数为 0xc0
    Write_Instruction( address); //写指令, 设置显示初始地址
}
```



```
//在第 X 行、第 Y 列开始显示, 指针 *s 所指向的字符串
void LCD_write_str(unsigned char X,unsigned char Y,unsigned char *s)
{
    LCD_SET_XY(X,Y);          //设置初始字符显示地址
    while( *s)                 //逐次写入显示字符, 直到最后一个字符"/0"
    {
        Write_Data( *s);      //写入当前字符并显示
        s++;                  //地址指针加 1, 指向下一个待写字符
    }
}

//在第 X 行、第 Y 列开始显示 Wdata 所对应的单个字符
void LCD_write_char(unsigned char X,unsigned char Y,unsigned char Wdata)
{
    LCD_SET_XY(X,Y);          //写地址
    Write_Data(Wdata);        //写入当前字符并显示
}
}
```

#### 4. 软件综合

商场灯光控制系统的软件综合如例 11.5 所示, 其中涉及的关于 PCF8563 的驱动函数代码可以参考前面两小节。应用代码在对 DS1302 初始化完成之后, 每隔 200ms 读取一次相应的时间数据, 并且对当前的小时时间进行比较, 然后控制继电器进行相应的动作。

##### 【例 11.5】商场灯光控制系统的软件综合

```
#include <iom16v.h>
#include <macros.h>
//μs 延时函数
void delay_us(unsigned int n)          //8×0.125=1μs
{
    int i,j;
    for(j=0;j<8;j++)
    {
        for(i=0;i<n;i++)
            NOP();
    }
}

//ms 延时函数
void delay_ms(unsigned int i)
{
    while(i--)
    {
        unsigned int j;
        for(j=1;j<=1332;j++)
        {
```

```

        ;
    }
}

//端口初始化
void port_init( void)
{
    PORTA = 0x00;
    DDRA = 0xFF;
    PORTB = 0x00;
    DDRB = 0xE0;
    PORTC = 0x00;
    DDRC = 0x07;
    PORTD = 0xC0;
    DDRD = 0xC0;
}

//ATmega16 初始化
void init_devices( void)
{
    CLI( ) ;
    port_init( ) ;
    MCUCR = 0x00;
    GICR = 0x00;
    TIMSK = 0x00;
    SEI( ) ;
}

//主函数
void main( void)
{
    unsigned char temp;
    init_devices( ) ;
    init_devices( ) ;
    LCD_init( ) ;                //LCD 初始化
    LCD_clear( ) ;
    ds1302_init( ) ;             //DS1302 初始化
    delay_ms( 10) ;
    while ( 1)
    {
        delay_ms( 200) ;        //每 200ms 更新一次时间
        ds1302_read_time( ) ;   //读取时间
        LCD_clear( ) ;          //清屏
        if( ( time_buf[ 4] ) > 22 || ( time_buf[ 4] < 7) )    //如果在 22 点和 7 点之间
    }
}

```



```

PORTD = 0x7F;           //继电器 1 闭合, 提供低电压
}
else
{
    PORTD = 0xBF;        //继电器 2 闭合, 提供高电压
}

temp = (time_buf[0] >> 4) + 0 ;
LCD_write_char(0, 0, temp);           /* 年 */
temp = (time_buf[0] & 0x0F) + 0 ;
LCD_write_char(1, 0, temp);
temp = (time_buf[1] >> 4) + 0 ;
LCD_write_char(2, 0, temp);
temp = (time_buf[1] & 0x0F) + 0 ;
LCD_write_char(3, 0, temp);
LCD_write_char(4, 0, ' ');

temp = (time_buf[2] >> 4) + 0 ;
LCD_write_char(5, 0, temp);           /* 月 */
temp = (time_buf[2] & 0x0F) + 0 ;
LCD_write_char(6, 0, temp);
LCD_write_char(7, 0, ' ');

temp = (time_buf[3] >> 4) + 0 ;
LCD_write_char(8, 0, temp);           /* 日 */
temp = (time_buf[3] & 0x0F) + 0 ;
LCD_write_char(9, 0, temp);
LCD_write_str(0, 1, "week:");
temp = (time_buf[7]) + 0 ;
LCD_write_char(5, 1, temp);           /* 周 */
temp = (time_buf[4] >> 4) + 0 ;
LCD_write_char(8, 1, temp);           /* 时 */
temp = (time_buf[4] & 0x0F) + 0 ;
LCD_write_char(9, 1, temp);
LCD_write_char(10, 1, ':');
temp = (time_buf[5] >> 4) + 0 ;
LCD_write_char(11, 1, temp);          /* 分 */
temp = (time_buf[5] & 0x0F) + 0 ;
LCD_write_char(12, 1, temp);
LCD_write_char(13, 1, ':');
temp = (time_buf[6] >> 4) + 0 ;
LCD_write_char(14, 1, temp);          /* 秒 */
temp = (time_buf[6] & 0x0F) + 0 ;
LCD_write_char(15, 1, temp);

```



### 11.3.5 应用系统的仿真和总结

在 Proteus 中绘制如图 11.22 的电路，其中涉及的典型 Proteus 器件如表 11.21 所示。

表 11.21 Proteus 电路器件列表

器件名称	库	子 库	说 明
ATmega16	Microprocessor ICs	AVR Family	ATmega16 单片机
RES	Resistors	Generic	通用电阻
CAP	Capacitors	Generic	电容
CAP - ELEC	Capacitors	Generic	极性电容
CRYSTAL	Miscellaneous	—	晶体
LM016L	Optoelectronics	Alphanumeric LCDs	1602 液晶模块
DS1302	Microprocessor ICs	Peripherals	时钟芯片
Relay	Switches & Relays	Relays (Generic)	继电器
OPOTCOUPLER - NPN	Optoelectronics	Optocouplers	光电隔离器

点击运行，可以看到当前的时间显示，并且当进入相应的时间之后会控制继电器自动切换，如图 11.33 所示。

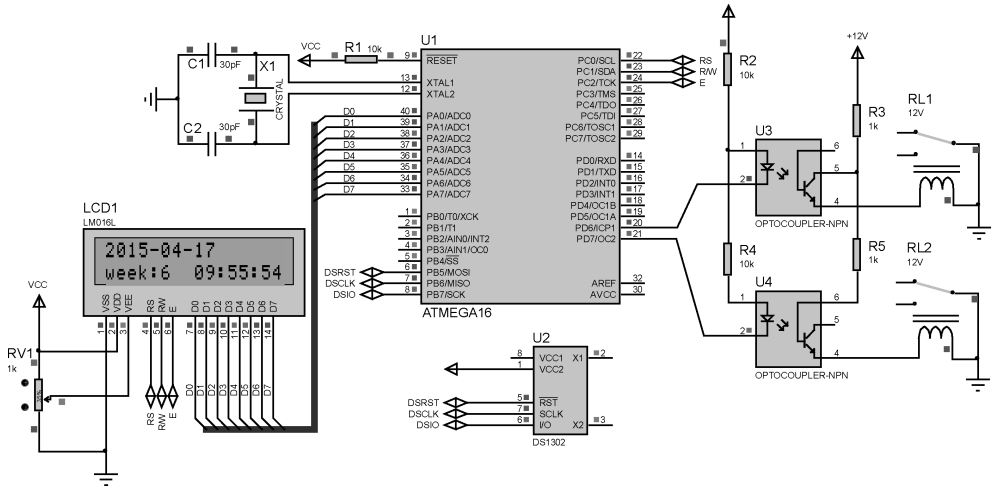


图 11.33 商场灯光控制系统的 Proteus 仿真



### 总结

在实际应用中，还应该加入相应的时间调整，手动设置切换时间等功能。